

CS471 Project 2

Generated by Doxygen 1.8.13

Contents

1	Namespace Index	1
1.1	Namespace List	1
2	Hierarchical Index	3
2.1	Class Hierarchy	3
3	Class Index	5
3.1	Class List	5
4	File Index	7
4.1	File List	7
5	Namespace Documentation	9
5.1	enums Namespace Reference	9
5.1.1	Enumeration Type Documentation	9
5.1.1.1	Algorithm	9
5.2	mdata Namespace Reference	10
5.3	mfunc Namespace Reference	10
5.3.1	Detailed Description	11
5.3.2	Typedef Documentation	11
5.3.2.1	mfuncPtr	11
5.3.3	Variable Documentation	11
5.3.3.1	NUM_FUNCTIONS	11
5.4	util Namespace Reference	11
5.4.1	Function Documentation	12
5.4.1.1	allocArray()	12
5.4.1.2	allocMatrix()	13
5.4.1.3	copyArray()	13
5.4.1.4	initArray()	14
5.4.1.5	initMatrix()	15
5.4.1.6	releaseArray()	15
5.4.1.7	releaseMatrix()	16

6	Class Documentation	17
6.1	enums::AlgorithmNames Struct Reference	17
6.1.1	Detailed Description	17
6.1.2	Member Function Documentation	17
6.1.2.1	get()	18
6.1.3	Member Data Documentation	18
6.1.3.1	BLIND_SEARCH	18
6.1.3.2	LOCAL_SEARCH	18
6.2	mdata::BlindSearch< T > Class Template Reference	19
6.2.1	Detailed Description	19
6.2.2	Member Function Documentation	20
6.2.2.1	run()	20
6.3	mdata::DataTable< T > Class Template Reference	21
6.3.1	Detailed Description	21
6.3.2	Constructor & Destructor Documentation	22
6.3.2.1	DataTable()	22
6.3.2.2	~DataTable()	22
6.3.3	Member Function Documentation	22
6.3.3.1	exportCSV()	23
6.3.3.2	getColLabel()	24
6.3.3.3	getEntry()	25
6.3.3.4	setColLabel()	25
6.3.3.5	setEntry()	26
6.4	mfunc::Experiment< T > Class Template Reference	26
6.4.1	Detailed Description	27
6.4.2	Constructor & Destructor Documentation	27
6.4.2.1	Experiment()	27
6.4.2.2	~Experiment()	28
6.4.3	Member Function Documentation	28
6.4.3.1	init()	28

6.4.3.2	testAllFunc()	30
6.4.3.3	testFuncThreaded()	32
6.5	mfunc::FunctionDesc Struct Reference	33
6.5.1	Detailed Description	33
6.5.2	Member Function Documentation	34
6.5.2.1	get()	34
6.6	mfunc::Functions< T > Struct Template Reference	35
6.6.1	Detailed Description	36
6.6.2	Member Function Documentation	36
6.6.2.1	ackleysOne()	36
6.6.2.2	ackleysTwo()	37
6.6.2.3	alpine()	37
6.6.2.4	dejong()	38
6.6.2.5	eggHolder()	39
6.6.2.6	exec()	39
6.6.2.7	get()	40
6.6.2.8	griewangk()	41
6.6.2.9	levy()	42
6.6.2.10	mastersCosineWave()	42
6.6.2.11	michalewicz()	43
6.6.2.12	nthroot()	44
6.6.2.13	pathological()	45
6.6.2.14	quartic()	46
6.6.2.15	rana()	46
6.6.2.16	rastrigin()	47
6.6.2.17	rosenbrok()	47
6.6.2.18	schwefel()	48
6.6.2.19	sineEnvelopeSineWave()	49
6.6.2.20	step()	49
6.6.2.21	stretchedVSineWave()	50

6.6.2.22	w()	50
6.7	util::IniReader Class Reference	51
6.7.1	Detailed Description	51
6.7.2	Constructor & Destructor Documentation	52
6.7.2.1	IniReader()	52
6.7.2.2	~IniReader()	52
6.7.3	Member Function Documentation	52
6.7.3.1	entryExists()	52
6.7.3.2	getEntry()	53
6.7.3.3	getEntryAs()	54
6.7.3.4	openFile()	54
6.7.3.5	sectionExists()	54
6.8	mdata::LocalSearch< T > Class Template Reference	55
6.8.1	Detailed Description	56
6.8.2	Member Function Documentation	56
6.8.2.1	run()	56
6.9	mdata::Population< T > Class Template Reference	58
6.9.1	Detailed Description	59
6.9.2	Constructor & Destructor Documentation	59
6.9.2.1	Population()	59
6.9.2.2	~Population()	60
6.9.3	Member Function Documentation	60
6.9.3.1	calcFitness()	60
6.9.3.2	generate()	61
6.9.3.3	getAllFitness()	62
6.9.3.4	getBestFitnessIndex()	62
6.9.3.5	getBestFitnessPtr()	63
6.9.3.6	getDimensionsSize()	64
6.9.3.7	getFitness()	65
6.9.3.8	getFitnessPtr()	66

6.9.3.9	getPopulationPtr()	66
6.9.3.10	getPopulationSize()	67
6.9.3.11	isReady()	67
6.9.3.12	outputFitness()	68
6.9.3.13	outputPopulation()	68
6.9.3.14	setFitness()	69
6.10	mfunc::RandomBounds< T > Struct Template Reference	70
6.10.1	Detailed Description	70
6.10.2	Member Data Documentation	70
6.10.2.1	max	70
6.10.2.2	min	71
6.11	mdata::SearchAlgorithm< T > Class Template Reference	71
6.11.1	Detailed Description	72
6.11.2	Constructor & Destructor Documentation	72
6.11.2.1	SearchAlgorithm()	72
6.11.2.2	~SearchAlgorithm()	72
6.11.3	Member Function Documentation	72
6.11.3.1	run()	73
6.11.3.2	startTimer()	73
6.11.3.3	stopTimer()	73
6.11.4	Member Data Documentation	73
6.11.4.1	timeDiff	74
6.11.4.2	timer	74
6.12	mdata::TestParameters< T > Struct Template Reference	74
6.12.1	Detailed Description	74
6.12.2	Constructor & Destructor Documentation	75
6.12.2.1	TestParameters()	75
6.12.3	Member Data Documentation	75
6.12.3.1	alg	75
6.12.3.2	alpha	76

6.12.3.3	<code>execTimesCol</code>	76
6.12.3.4	<code>execTimesRow</code>	76
6.12.3.5	<code>execTimesTable</code>	76
6.12.3.6	<code>funcId</code>	77
6.12.3.7	<code>resultsCol</code>	77
6.12.3.8	<code>resultsRow</code>	77
6.12.3.9	<code>resultsTable</code>	77
6.13	<code>mdata::TestResult< T ></code> Struct Template Reference	78
6.13.1	Detailed Description	78
6.13.2	Constructor & Destructor Documentation	78
6.13.2.1	<code>TestResult()</code>	78
6.13.3	Member Data Documentation	78
6.13.3.1	<code>err</code>	79
6.13.3.2	<code>execTime</code>	79
6.13.3.3	<code>fitness</code>	79
6.14	<code>ThreadPool</code> Class Reference	79
6.14.1	Detailed Description	80
6.14.2	Constructor & Destructor Documentation	80
6.14.2.1	<code>ThreadPool()</code>	80
6.14.2.2	<code>~ThreadPool()</code>	81
6.14.3	Member Function Documentation	81
6.14.3.1	<code>enqueue()</code>	81
6.14.3.2	<code>stopAndJoinAll()</code>	81

7 File Documentation	83
7.1 include/blindsearch.h File Reference	83
7.1.1 Detailed Description	84
7.2 blindsearch.h	85
7.3 include/datatable.h File Reference	85
7.3.1 Detailed Description	87
7.4 datatable.h	87
7.5 include/experiment.h File Reference	88
7.5.1 Detailed Description	90
7.6 experiment.h	90
7.7 include/inireader.h File Reference	91
7.7.1 Detailed Description	92
7.8 inireader.h	93
7.9 include/localsearch.h File Reference	93
7.9.1 Detailed Description	95
7.9.2 Macro Definition Documentation	95
7.9.2.1 DEC_PRECISION	95
7.10 localsearch.h	95
7.11 include/mem.h File Reference	97
7.11.1 Detailed Description	99
7.12 mem.h	99
7.13 include/mfuncptr.h File Reference	100
7.13.1 Detailed Description	102
7.14 mfuncptr.h	102
7.15 include/mfunctions.h File Reference	102
7.15.1 Detailed Description	104
7.15.2 Macro Definition Documentation	105
7.15.2.1 _ackleysOneDesc	105
7.15.2.2 _ackleysTwoDesc	105
7.15.2.3 _alpineDesc	105

7.15.2.4	_dejongDesc	105
7.15.2.5	_eggHolderDesc	105
7.15.2.6	_griewangkDesc	106
7.15.2.7	_levyDesc	106
7.15.2.8	_mastersCosineWaveDesc	106
7.15.2.9	_michalewiczDesc	106
7.15.2.10	_pathologicalDesc	106
7.15.2.11	_quarticDesc	107
7.15.2.12	_ranaDesc	107
7.15.2.13	_rastriginDesc	107
7.15.2.14	_rosenbrokDesc	107
7.15.2.15	_schwefelDesc	107
7.15.2.16	_sineEnvelopeSineWaveDesc	108
7.15.2.17	_stepDesc	108
7.15.2.18	_stretchedVSineWaveDesc	108
7.15.2.19	_USE_MATH_DEFINES	108
7.16	mfunctions.h	109
7.17	include/population.h File Reference	114
7.17.1	Detailed Description	116
7.18	population.h	116
7.19	include/searchalg.h File Reference	117
7.19.1	Detailed Description	118
7.20	searchalg.h	119
7.21	include/stringutils.h File Reference	120
7.21.1	Detailed Description	121
7.22	stringutils.h	121
7.23	include/testparam.h File Reference	122
7.23.1	Detailed Description	123
7.24	testparam.h	123
7.25	include/testresult.h File Reference	124

7.25.1 Detailed Description	124
7.26 testresult.h	125
7.27 include/threadpool.h File Reference	125
7.28 threadpool.h	126
7.29 src/experiment.cpp File Reference	127
7.29.1 Macro Definition Documentation	128
7.29.1.1 INI_FUNC_RANGE_SECTION	128
7.29.1.2 INI_TEST_ALGORITHM	128
7.29.1.3 INI_TEST_ALPHA	129
7.29.1.4 INI_TEST_DIMENSIONS	129
7.29.1.5 INI_TEST_EXECTIMESFILE	129
7.29.1.6 INI_TEST_ITERATIONS	129
7.29.1.7 INI_TEST_NUMTHREADS	129
7.29.1.8 INI_TEST_POPULATION	130
7.29.1.9 INI_TEST_RESULTSFILE	130
7.29.1.10 INI_TEST_SECTION	130
7.30 experiment.cpp	130
7.31 src/inireader.cpp File Reference	136
7.31.1 Detailed Description	137
7.32 inireader.cpp	137
7.33 src/main.cpp File Reference	139
7.33.1 Detailed Description	139
7.33.2 Function Documentation	140
7.33.2.1 main()	140
7.33.2.2 runExp()	140
7.34 main.cpp	141
7.35 src/population.cpp File Reference	142
7.35.1 Detailed Description	142
7.36 population.cpp	143

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

enums	9
mdata	10
mfunc	10
util	11

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

enums::AlgorithmNames	17
mdata::DataTable< T >	21
mfunc::Experiment< T >	26
mfunc::FunctionDesc	33
mfunc::Functions< T >	35
util::IniReader	51
mdata::Population< T >	58
mfunc::RandomBounds< T >	70
mdata::SearchAlgorithm< T >	71
mdata::BlindSearch< T >	19
mdata::LocalSearch< T >	55
mdata::TestParameters< T >	74
mdata::TestResult< T >	78
ThreadPool	79

Chapter 3

Class Index

3.1 Class List

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

enums::AlgorithmNames	
Struct that contains constant string names for the different search algorithms	17
mdata::BlindSearch< T >	
The BlindSearch class implements the Blind Search algorithm, which is ran using the overridden SearchAlgorithm::run() function	19
mdata::DataTable< T >	
The DataTable class is a simple table of values with labeled columns	21
mfunc::Experiment< T >	
Contains classes for running the CS471 project experiment	26
mfunc::FunctionDesc	
Get() returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null	33
mfunc::Functions< T >	
Struct containing all static math functions. A function can be called directly by name, or indirectly using Functions::get or Functions::exec	35
util::IniReader	
Simple *.ini file reader and parser	51
mdata::LocalSearch< T >	
The LocalSearch class implements the Local Search algorithm, which is ran using the overridden SearchAlgorithm::run() function	55
mdata::Population< T >	
Data class for storing a multi-dimensional population of data with the associated fitness	58
mfunc::RandomBounds< T >	
Simple struct for storing the minimum and maximum input vector bounds for a function	70
mdata::SearchAlgorithm< T >	
The SearchAlgorithm class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm	71
mdata::TestParameters< T >	
Packs together various test experiment parameters	74
mdata::TestResult< T >	
.	78
ThreadPool	
.	79

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/ blindsearch.h	Implements the BlindSearch class, which inherits SearchAlgorithm. BlindSearch::run executes the blind search algorithm on a given population	83
include/ datatable.h	Header file for the DataTable class, which represents a spreadsheet/table of values that can easily be exported to a *.csv file	85
include/ experiment.h	Header file for the Experiment class. Contains the basic logic and functions to run the cs471 project experiment	88
include/ inireader.h	Header file for the IniReader class, which can open and parse simple *.ini files	91
include/ localsearch.h	93
include/ mem.h	Header file for various memory utility functions	97
include/ mfuncptr.h	Contains the type definition for mfuncPtr, a templated function pointer to one of the math functions in mfunctions.h	100
include/ mfunctions.h	Contains various math function definitions	102
include/ population.h	Header file for the Population class. Stores a population and resulting fitness values	114
include/ searchalg.h	Defines the SearchAlgorithm class, Algorithm enum, and AlgorithmNames struct. The SearchAlgorithm class serves as a base class for implemented search algorithms	117
include/ stringutils.h	Contains various string manipulation helper functions	120
include/ testparam.h	Contains the definition of the TestParameters struct, which is a data type used to transfer test parameters between functions	122
include/ testresult.h	Simple structure that packs together various return values for the search algorithms. functions	124
include/ threadpool.h	125
src/ experiment.cpp	127
src/ inireader.cpp	Implementation file for the IniReader class, which can open and parse simple *.ini files	136

src/main.cpp	
Program entry point. Creates and runs CS471 project 2 experiment	139
src/population.cpp	
Implementation file for the Population class. Stores a population and fitness values	142

Chapter 5

Namespace Documentation

5.1 enums Namespace Reference

Classes

- struct [AlgorithmNames](#)
Struct that contains constant string names for the different search algorithms.

Enumerations

- enum [Algorithm](#) { [BlindSearch](#) = 0, [LocalSearch](#) = 1, [Count](#) = 2 }
Enum of different available search algorithms.

5.1.1 Enumeration Type Documentation

5.1.1.1 Algorithm

enum [enums::Algorithm](#)

Enum of different available search algorithms.

Enumerator

BlindSearch	
LocalSearch	
Count	

Definition at line [28](#) of file [searchalg.h](#).

00029 {

```

00030         BlindSearch = 0,
00031         LocalSearch = 1,
00032         Count = 2
00033     };

```

5.2 mdata Namespace Reference

Classes

- class [BlindSearch](#)
The [BlindSearch](#) class implements the Blind Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.
- class [DataTable](#)
The [DataTable](#) class is a simple table of values with labeled columns.
- class [LocalSearch](#)
The [LocalSearch](#) class implements the Local Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.
- class [Population](#)
Data class for storing a multi-dimensional population of data with the associated fitness.
- class [SearchAlgorithm](#)
The [SearchAlgorithm](#) class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm.
- struct [TestParameters](#)
Packs together various test experiment parameters.
- struct [TestResult](#)

5.3 mfunc Namespace Reference

Classes

- class [Experiment](#)
Contains classes for running the CS471 project experiment.
- struct [FunctionDesc](#)
[get\(\)](#) returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null
- struct [Functions](#)
Struct containing all static math functions. A function can be called directly by name, or indirectly using [Functions::get](#) or [Functions::exec](#).
- struct [RandomBounds](#)
Simple struct for storing the minimum and maximum input vector bounds for a function.

Typedefs

- template<class T >
using [mfuncPtr](#) = T(*)(T *, size_t)
Function pointer that takes two arguments T* and size_t, and returns a T value.

Variables

- constexpr const unsigned int [NUM_FUNCTIONS](#) = 18

5.3.1 Detailed Description

Scope for all math functions

5.3.2 Typedef Documentation

5.3.2.1 mfuncPtr

```
template<class T >
using mfunc::mfuncPtr = typedef T (*)(T*, size_t)
```

Function pointer that takes two arguments T* and size_t, and returns a T value.

Template Parameters

<i>T</i>	Data type for vector and return value
----------	---------------------------------------

Definition at line 28 of file [mfuncptr.h](#).

5.3.3 Variable Documentation

5.3.3.1 NUM_FUNCTIONS

```
constexpr const unsigned int mfunc::NUM_FUNCTIONS = 18
```

Constant value for the total number of math functions contained in this namespace

Definition at line 47 of file [mfunctions.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

5.4 util Namespace Reference

Classes

- class [IniReader](#)
The [IniReader](#) class is a simple *.ini file reader and parser.

Functions

- `template<class T = double>`
`void initArray (T *a, size_t size, T val)`
Initializes an array with some set value.
- `template<class T = double>`
`void initMatrix (T **m, size_t rows, size_t cols, T val)`
Initializes a matrix with a set value for each entry.
- `template<class T = double>`
`void releaseArray (T *&a)`
Releases an allocated array's memory and sets the pointer to nullptr.
- `template<class T = double>`
`void releaseMatrix (T **&m, size_t rows)`
Releases an allocated matrix's memory and sets the pointer to nullptr.
- `template<class T = double>`
`T * allocArray (size_t size)`
Allocates a new array of the given data type.
- `template<class T = double>`
`T ** allocMatrix (size_t rows, size_t cols)`
Allocates a new matrix of the given data type.
- `template<class T = double>`
`void copyArray (T *src, T *dest, size_t size)`
Copies the elements from one equal-sized array to another.

5.4.1 Function Documentation

5.4.1.1 `allocArray()`

```
template<class T = double>
T* util::allocArray (
    size_t size ) [inline]
```

Allocates a new array of the given data type.

Template Parameters

<i>Data</i>	type of the array
-------------	-------------------

Parameters

<i>size</i>	Number of elements in the array
-------------	---------------------------------

Returns

Returns a pointer to the new array, or nullptr allocation fails

Definition at line [108](#) of file [mem.h](#).


```

00109      {
00110          return new(std::nothrow) T[size];
00111      }

```

5.4.1.2 allocMatrix()

```

template<class T = double>
T** util::allocMatrix (
    size_t rows,
    size_t cols ) [inline]

```

Allocates a new matrix of the given data type.

Template Parameters

<i>Data</i>	type of the matrix entries
-------------	----------------------------

Parameters

<i>rows</i>	The number of rows
<i>cols</i>	The number of columns

Returns

Returns a pointer to the new matrix, or nullptr if allocation fails

Definition at line 122 of file [mem.h](#).

```

00123      {
00124          T** m = (T**)allocArray<T*>(rows);
00125          if (m == nullptr) return nullptr;
00126
00127          for (size_t i = 0; i < rows; i++)
00128          {
00129              m[i] = allocArray<T>(cols);
00130              if (m[i] == nullptr)
00131              {
00132                  releaseMatrix<T>(m, rows);
00133                  return nullptr;
00134              }
00135          }
00136
00137          return m;
00138      }

```

5.4.1.3 copyArray()

```

template<class T = double>
void util::copyArray (
    T * src,
    T * dest,
    size_t size ) [inline]

```

Copies the elements from one equal-sized array to another.

Template Parameters

<i>Data</i>	type of the array
-------------	-------------------

Parameters

<i>src</i>	Source array from where the elements will be copied from
<i>dest</i>	Destination array from where the elements will be copied to
<i>size</i>	Number of elements in the array

Definition at line 149 of file [mem.h](#).

```

00150     {
00151         for (size_t i = 0; i < size; i++)
00152             dest[i] = src[i];
00153     }
```

5.4.1.4 initArray()

```

template<class T = double>
void util::initArray (
    T * a,
    size_t size,
    T val ) [inline]
```

Initializes an array with some set value.

Template Parameters

<i>Data</i>	type of array
-------------	---------------

Parameters

<i>a</i>	Pointer to array
<i>size</i>	Size of the array
<i>val</i>	Value to initialize the array to

Definition at line 29 of file [mem.h](#).

Referenced by [initMatrix\(\)](#).

```

00030     {
00031         if (a == nullptr) return;
00032
00033         for (size_t i = 0; i < size; i++)
00034         {
00035             a[i] = val;
00036         }
00037     }
```

5.4.1.5 initMatrix()

```
template<class T = double>
void util::initMatrix (
    T ** m,
    size_t rows,
    size_t cols,
    T val ) [inline]
```

Initializes a matrix with a set value for each entry.

Template Parameters

Data	type of matrix entries
------	------------------------

Parameters

<i>m</i>	Pointer to a matrix
<i>rows</i>	Number of rows in matrix
<i>cols</i>	Number of columns in matrix
<i>val</i>	Value to initialize the matrix to

Definition at line 49 of file [mem.h](#).

References [initArray\(\)](#).

```
00050     {
00051         if (m == nullptr) return;
00052         for (size_t i = 0; i < rows; i++)
00053         {
00054             initArray(m[i], cols, val);
00055         }
00056     }
00057 }
```

5.4.1.6 releaseArray()

```
template<class T = double>
void util::releaseArray (
    T *& a )
```

Releases an allocated array's memory and sets the pointer to nullptr.

Template Parameters

Data	type of array
------	---------------

Parameters

<i>a</i>	Pointer to array
----------	------------------

Definition at line 66 of file [mem.h](#).

```

00067     {
00068         if (a == nullptr) return;
00069
00070         delete[] a;
00071         a = nullptr;
00072     }

```

5.4.1.7 releaseMatrix()

```

template<class T = double>
void util::releaseMatrix (
    T **& m,
    size_t rows )

```

Releases an allocated matrix's memory and sets the pointer to nullptr.

Template Parameters

<i>Data</i>	type of the matrix
-------------	--------------------

Parameters

<i>m</i>	Pointer th the matrix
<i>rows</i>	The number of rows in the matrix

Definition at line 82 of file [mem.h](#).

Referenced by [mdata::DataTable< T >::~~DataTable\(\)](#).

```

00083     {
00084         if (m == nullptr) return;
00085
00086         for (size_t i = 0; i < rows; i++)
00087         {
00088             if (m[i] != nullptr)
00089             {
00090                 // Release each row
00091                 releaseArray<T>(m[i]);
00092             }
00093         }
00094
00095         // Release columns
00096         delete[] m;
00097         m = nullptr;
00098     }

```

Chapter 6

Class Documentation

6.1 `enums::AlgorithmNames` Struct Reference

Struct that contains constant string names for the different search algorithms.

```
#include <searchalg.h>
```

Static Public Member Functions

- static const char * [get](#) ([Algorithm](#) alg)

Static Public Attributes

- static constexpr const char * [BLIND_SEARCH](#) = "Blind Search"
- static constexpr const char * [LOCAL_SEARCH](#) = "Local Search"

6.1.1 Detailed Description

Struct that contains constant string names for the different search algorithms.

Definition at line [39](#) of file [searchalg.h](#).

6.1.2 Member Function Documentation

6.1.2.1 get()

```
static const char* enums::AlgorithmNames::get (  
    Algorithm alg ) [inline], [static]
```

Definition at line 44 of file [searchalg.h](#).

References [enums::BlindSearch](#), and [enums::LocalSearch](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

```
00045     {  
00046         switch (alg)  
00047         {  
00048             case Algorithm::BlindSearch:  
00049                 return BLIND_SEARCH;  
00050             case Algorithm::LocalSearch:  
00051                 return LOCAL_SEARCH;  
00052             default:  
00053                 return "";  
00054                 break;  
00055         }  
00056     }
```

6.1.3 Member Data Documentation

6.1.3.1 BLIND_SEARCH

```
constexpr const char* enums::AlgorithmNames::BLIND_SEARCH = "Blind Search" [static]
```

Definition at line 41 of file [searchalg.h](#).

6.1.3.2 LOCAL_SEARCH

```
constexpr const char* enums::AlgorithmNames::LOCAL_SEARCH = "Local Search" [static]
```

Definition at line 42 of file [searchalg.h](#).

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

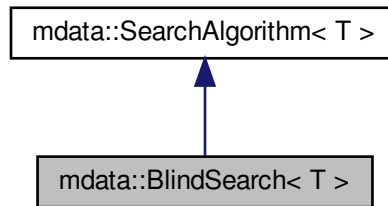
- [include/searchalg.h](#)

6.2 mdata::BlindSearch< T > Class Template Reference

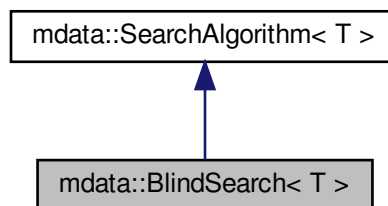
The [BlindSearch](#) class implements the Blind Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.

```
#include <blindsearch.h>
```

Inheritance diagram for mdata::BlindSearch< T >:



Collaboration diagram for mdata::BlindSearch< T >:



Public Member Functions

- virtual [TestResult< T >](#) [run](#) ([mfunc::mfuncPtr< T >](#) funcPtr, const T fMin, const T fMax, [Population< T >](#) *const pop, const T alpha)
Executes Blind Search with the given population and parameters.

Additional Inherited Members

6.2.1 Detailed Description

```
template<class T>
class mdata::BlindSearch< T >
```

The [BlindSearch](#) class implements the Blind Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.

Template Parameters

<i>T</i>	Data type used
----------	----------------

Definition at line 27 of file [blindsearch.h](#).

6.2.2 Member Function Documentation

6.2.2.1 run()

```
template<class T >
virtual TestResult<T> mdata::BlindSearch< T >::run (
    mfunc::mfuncPtr< T > funcPtr,
    const T fMin,
    const T fMax,
    Population< T > *const pop,
    const T alpha ) [inline], [virtual]
```

Executes Blind Search with the given population and parameters.

Parameters

<i>funcPtr</i>	Function pointer to the math function being used to generate the population
<i>fMin</i>	Minimum bound for the population matrix vector components
<i>fMax</i>	Maximum bound for the population matrix vector components
<i>pop</i>	Pointer to a population object that will be used in the blind search
<i>alpha</i>	Unused in this algorithm

Returns

TestResult<T> Returns a [TestResult](#) struct containing the error code, fitness, and execution time

Implements [mdata::SearchAlgorithm< T >](#).

Definition at line 44 of file [blindsearch.h](#).

References [mdata::Population< T >::calcFitness\(\)](#), [mdata::Population< T >::generate\(\)](#), [mdata::Population< T >::getBestFitnessPtr\(\)](#), [mdata::Population< T >::getDimensionsSize\(\)](#), [mdata::Population< T >::getPopulationSize\(\)](#), [mdata::SearchAlgorithm< T >::startTimer\(\)](#), and [mdata::SearchAlgorithm< T >::stopTimer\(\)](#).

```
00045     {
00046         // Get population size and dimensions
00047         size_t popSize = pop->getPopulationSize();
00048         size_t dimSize = pop->getDimensionsSize();
00049
00050         // Make sure funcPtr is valid;
00051         if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with
error code 1
00052
00053         // Start recording execution time
00054         startTimer();
```



```

00055
00056         // Generate values for population vector matrix
00057         pop->generate(fMin, fMax);
00058
00059         // For each population vector, calculate the fitness using the funcPtr
00060         for (size_t sol = 0; sol < popSize; sol++)
00061         {
00062             // Populate fitness values using given math function pointer
00063             if (!pop->calcFitness(sol, funcPtr))
00064                 return TResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
00065         }
00066
00067         // Return best fitness value in population
00068         return TResult<T>(0, *pop->getBestFitnessPtr(), stopTimer());
00069     }

```

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

- include/[blindsearch.h](#)

6.3 mdata::DataTable< T > Class Template Reference

The [DataTable](#) class is a simple table of values with labeled columns.

```
#include <datatable.h>
```

Public Member Functions

- [DataTable](#) (size_t _rows, size_t _cols)
Construct a new Data Table object Throws std::length_error and std::bad_alloc.
- [~DataTable](#) ()
Destroy the Data Table object.
- std::string [getColLabel](#) (size_t colIndex)
Gets the string label for the column with the given index.
- void [setColLabel](#) (size_t colIndex, std::string newLabel)
Sets the string label for the column with the given index.
- T [getEntry](#) (size_t row, size_t col)
Returns the value in the table at the given row and column.
- void [setEntry](#) (size_t row, size_t col, T val)
Set the value for the table entry at the given row and column.
- bool [exportCSV](#) (const char *filePath)
Exports the contents of this [DataTable](#) to a .csv file.

6.3.1 Detailed Description

```
template<class T>
class mdata::DataTable< T >
```

The [DataTable](#) class is a simple table of values with labeled columns.

– Initialize a [DataTable](#) object with a specified number of rows and columns: [DataTable](#) table(rows, columns);

Set a column's label:

```
table.setColLabel(0, "Column 1");
```

Set an entry in the table:

```
table.setEntry(n, m, value);
```

Where 'n' is the row, 'm' is the column, and 'value' is the value of the entry

Export the table to a *.csv file:

```
bool success = table.exportCSV("my_file.csv");
```

Definition at line 50 of file [datatable.h](#).

6.3.2 Constructor & Destructor Documentation

6.3.2.1 DataTable()

```
template<class T>
mdata::DataTable< T >::DataTable (
    size_t _rows,
    size_t _cols ) [inline]
```

Construct a new Data Table object Throws `std::length_error` and `std::bad_alloc`.

Parameters

<code>_rows</code>	Number of rows in table
<code>_cols</code>	Number of columns in table

Definition at line 60 of file [datatable.h](#).

```
00060                                     : rows(_rows), cols(_cols), dataMatrix(nullptr)
00061     {
00062         if (rows == 0)
00063             throw std::length_error("Table rows must be greater than 0.");
00064         else if (cols == 0)
00065             throw std::length_error("Table columns must be greater than 0.");
00066
00067         dataMatrix = util::allocMatrix<T>(rows, cols);
00068         if (dataMatrix == nullptr)
00069             throw std::bad_alloc();
00070
00071         colLabels.resize(_cols, std::string());
00072     }
```

6.3.2.2 ~DataTable()

```
template<class T>
mdata::DataTable< T >::~DataTable ( ) [inline]
```

Destroy the Data Table object.

Definition at line 77 of file [datatable.h](#).

References [util::releaseMatrix\(\)](#).

```
00078     {
00079         util::releaseMatrix(dataMatrix, rows);
00080     }
```

6.3.3 Member Function Documentation

6.3.3.1 exportCSV()

```
template<class T>
bool mdata::DataTable< T >::exportCSV (
    const char * filePath ) [inline]
```

Exports the contents of this [DataTable](#) to a .csv file.

Parameters

<i>filePath</i>	Path to the file that will be filled with this table's values
-----------------	---

Returns

true If the file was successfully written to
false If there was an error opening the file

Definition at line 155 of file [datatable.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#).

```

00156         {
00157             if (dataMatrix == nullptr) return false;
00158
00159             using namespace std;
00160             ofstream outFile;
00161             outFile.open(filePath, ofstream::out | ofstream::trunc);
00162             if (!outFile.good()) return false;
00163
00164             // Print column labels
00165             for (unsigned int c = 0; c < cols; c++)
00166             {
00167                 outFile << colLabels[c];
00168                 if (c < cols - 1) outFile << ",";
00169             }
00170
00171             outFile << endl;
00172
00173             // Print data rows
00174             for (unsigned int r = 0; r < rows; r++)
00175             {
00176                 for (unsigned int c = 0; c < cols; c++)
00177                 {
00178                     outFile << std::setprecision(8) << dataMatrix[r][c];
00179                     if (c < cols - 1) outFile << ",";
00180                 }
00181                 outFile << endl;
00182             }
00183
00184             outFile.close();
00185             return true;
00186         }

```

6.3.3.2 getColLabel()

```

template<class T>
std::string mdata::DataTable< T >::getColLabel (
    size_t colIndex ) [inline]

```

Gets the string label for the column with the given index.

Parameters

<i>colIndex</i>	Index of the column
-----------------	---------------------

Returns

std::string String value of the column label

Definition at line 88 of file [datatable.h](#).

```

00089     {
00090         if (colIndex >= colLabels.size())
00091             throw std::out_of_range("Column index out of range");
00092
00093         return colLabels[colIndex];
00094     }

```

6.3.3.3 getEntry()

```

template<class T>
T mdata::DataTable< T >::getEntry (
    size_t row,
    size_t col ) [inline]

```

Returns the value in the table at the given row and column.

Parameters

<i>row</i>	Row index of the table
<i>col</i>	Column index of the table

Returns

T Value of the entry at the given row and column

Definition at line 117 of file [datatable.h](#).

```

00118     {
00119         if (dataMatrix == nullptr)
00120             throw std::runtime_error("Data matrix not allocated");
00121         if (row >= rows)
00122             throw std::out_of_range("Table row out of range");
00123         else if (col >= cols)
00124             throw std::out_of_range("Table column out of range");
00125
00126         return dataMatrix[row][col];
00127     }

```

6.3.3.4 setColLabel()

```

template<class T>
void mdata::DataTable< T >::setColLabel (
    size_t colIndex,
    std::string newLabel ) [inline]

```

Sets the string label for the column with the given index.

Parameters

<i>colIndex</i>	Index of the column
<i>newLabel</i>	New string label for the column

Definition at line 102 of file [datatable.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#).

```

00103     {
00104         if (colIndex >= colLabels.size())
00105             throw std::out_of_range("Column index out of range");
00106
00107         colLabels[colIndex] = newLabel;
00108     }

```

6.3.3.5 setEntry()

```

template<class T>
void mdata::DataTable< T >::setEntry (
    size_t row,
    size_t col,
    T val ) [inline]

```

Set the value for the table entry at the given row and column.

Parameters

<i>row</i>	Row index of the table
<i>col</i>	Column index of the table
<i>val</i>	New value for the entry

Definition at line 136 of file [datatable.h](#).

```

00137     {
00138         if (dataMatrix == nullptr)
00139             throw std::runtime_error("Data matrix not allocated");
00140         if (row >= rows)
00141             throw std::out_of_range("Table row out of range");
00142         else if (col >= cols)
00143             throw std::out_of_range("Table column out of range");
00144
00145         dataMatrix[row][col] = val;
00146     }

```

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

- [include/datatable.h](#)

6.4 mfunc::Experiment< T > Class Template Reference

Contains classes for running the CS471 project experiment.

```
#include <experiment.h>
```

Public Member Functions

- [Experiment](#) ()
Construct a new [Experiment](#) object.
- [~Experiment](#) ()
Destroys the [Experiment](#) object.
- bool [init](#) (const char *paramFile)
Initializes the CS471 project 2 experiment. Opens the given parameter file and extracts test parameters. Allocates memory for function vectors and function bounds. Extracts all function bounds.
- int [testAllFunc](#) ()
*Executes all functions as specified in the CS471 project 2 document, records results, and outputs the data as a *.csv file.*
- int [testFuncThreaded](#) (mdata::TestParameters< T > tParams)
Executes a single iteration of a test with the given parameters.

6.4.1 Detailed Description

```
template<class T>
class mfunc::Experiment< T >
```

Contains classes for running the CS471 project experiment.

The [Experiment](#) class opens a given parameter .ini file and executes the CS471 project 2 experiment with the specified parameters. [runAllFunc\(\)](#) runs all 18 functions defined in [mfunctions.h](#) a given number of times with vectors of random values that have a given number of dimensions and collects all results/data. This data is then entered into a DataTable and exported as a *.csv file.

Definition at line 52 of file [experiment.h](#).

6.4.2 Constructor & Destructor Documentation

6.4.2.1 Experiment()

```
template<class T >
Experiment::Experiment ( )
```

Construct a new [Experiment](#) object.

Definition at line 43 of file [experiment.cpp](#).

```
00044      : vBounds(nullptr), tPool(nullptr), resultsFile(""), execTimesFile(""), iterations(0)
00045  {
00046  }
```

6.4.2.2 ~Experiment()

```
template<class T >
Experiment::~~Experiment ( )
```

Destroys the [Experiment](#) object.

Definition at line 53 of file [experiment.cpp](#).

```
00054 {
00055     releaseThreadPool();
00056     releasePopulationPool();
00057     releaseVBounds();
00058 }
```

6.4.3 Member Function Documentation

6.4.3.1 init()

```
template<class T >
bool Experiment::init (
    const char * paramFile )
```

Initializes the CS471 project 2 experiment. Opens the given parameter file and extracts test parameters. Allocates memory for function vectors and function bounds. Extracts all function bounds.

Parameters

<i>paramFile</i>	File path to the parameter ini file
------------------	-------------------------------------

Returns

Returns true if initialization was successful. Otherwise false.

Definition at line 69 of file [experiment.cpp](#).

References [enums::Count](#), [enums::AlgorithmNames::get\(\)](#), [util::IniReader::getEntry\(\)](#), [util::IniReader::getEntryAs\(\)](#), [INI_TEST_ALGORITHM](#), [INI_TEST_ALPHA](#), [INI_TEST_DIMENSIONS](#), [INI_TEST_EXECTIMESFILE](#), [INI_TEST_ITERATIONS](#), [INI_TEST_NUMTHREADS](#), [INI_TEST_POPULATION](#), [INI_TEST_RESULTSFILE](#), [INI_TEST_SECTION](#), and [util::IniReader::openFile\(\)](#).

Referenced by [runExp\(\)](#).

```
00070 {
00071     try
00072     {
00073         // Open and parse parameters file
00074         if (!iniParams.openFile(paramFile))
00075         {
00076             cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
00077             return false;
00078         }
00079     }
```



```

00080         // Extract test parameters from ini file
00081         long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION,
INI_TEST_POPULATION);
00082         long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION,
INI_TEST_DIMENSIONS);
00083         long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION,
INI_TEST_ITERATIONS);
00084         long numberThreads = iniParams.getEntryAs<long>(
INI_TEST_SECTION, INI_TEST_NUMTHREADS);
00085         alpha = iniParams.getEntryAs<T>(INI_TEST_SECTION,
INI_TEST_ALPHA);
00086         unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(
INI_TEST_SECTION, INI_TEST_ALGORITHM);
00087         resultsFile = iniParams.getEntry(INI_TEST_SECTION,
INI_TEST_RESULTSFILE);
00088         execTimesFile = iniParams.getEntry(INI_TEST_SECTION,
INI_TEST_EXECTIMESFILE);
00089
00090         // Verify test parameters
00091         if (numberSol <= 0)
00092         {
00093             cerr << "Experiment init failed: Param file [test]->"
00094                 << INI_TEST_POPULATION << " entry missing or out of bounds: " <<
paramFile << endl;
00095             return false;
00096         }
00097         else if (numberDim <= 0)
00098         {
00099             cerr << "Experiment init failed: Param file [test]->"
00100                 << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " <<
paramFile << endl;
00101             return false;
00102         }
00103         else if (numberIter <= 0)
00104         {
00105             cerr << "Experiment init failed: Param file [test]->"
00106                 << INI_TEST_ITERATIONS << " entry missing or out of bounds: " <<
paramFile << endl;
00107             return false;
00108         }
00109         else if (numberThreads <= 0)
00110         {
00111             cerr << "Experiment init failed: Param file [test]->"
00112                 << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " <<
paramFile << endl;
00113             return false;
00114         }
00115         else if (alpha == 0)
00116         {
00117             cerr << "Experiment init failed: Param file [test]->"
00118                 << INI_TEST_ALPHA << " is missing or is equal to zero: " << paramFile << endl;
00119             return false;
00120         }
00121         else if (selectedAlg >= static_cast<unsigned int>(
enums::Algorithm::Count))
00122         {
00123             cerr << "Experiment init failed: Param file [test]->"
00124                 << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile
<< endl;
00125             return false;
00126         }
00127
00128         // Cast iterations and test algorithm to correct types
00129         iterations = (size_t)numberIter;
00130         testAlg = static_cast<enums::Algorithm>(selectedAlg);
00131
00132         // Print test parameters to console
00133         cout << "Population size: " << numberSol << endl;
00134         cout << "Dimensions: " << numberDim << endl;
00135         cout << "Iterations: " << iterations << endl;
00136         cout << "Alpha value: " << alpha << endl;
00137         cout << "Algorithm: " << enums::AlgorithmNames::get(testAlg) << endl;
00138
00139         // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
00140         if (!allocatePopulationPool((size_t)numberThreads, (size_t)numberSol, (size_t)numberDim))
00141         {
00142             cerr << "Experiment init failed: Unable to allocate populations." << endl;
00143             return false;
00144         }
00145
00146         // Allocate memory for function vector bounds
00147         if (!allocateVBounds())
00148         {
00149             cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
00150             return false;
00151         }

```

```

00152
00153     // Fill function bounds array with data parsed from iniParams
00154     if (!parseFuncBounds())
00155     {
00156         cerr << "Experiment init failed: Unable to parse vector bounds array." << endl;
00157         return false;
00158     }
00159
00160     // Allocate thread pool
00161     if (!allocateThreadPool((size_t)numberThreads))
00162     {
00163         cerr << "Experiment init failed: Unable to allocate thread pool." << endl;
00164         return false;
00165     }
00166
00167     cout << "Started " << numberThreads << " worker threads ..." << endl;
00168
00169     // Ready to run an experiment
00170     return true;
00171 }
00172 catch (const std::exception& ex)
00173 {
00174     cerr << "Exception occurred while initializing experiment: " << ex.what() << endl;
00175     return false;
00176 }
00177 catch (...)
00178 {
00179     cerr << "Unknown Exception occurred while initializing experiment." << endl;
00180     return false;
00181 }
00182 }

```

6.4.3.2 testAllFunc()

```

template<class T >
int Experiment::testAllFunc ( )

```

Executes all functions as specified in the CS471 project 2 document, records results, and outputs the data as a *.csv file.

Returns

Returns 0 on success. Returns a non-zero error code on failure.

Definition at line 191 of file [experiment.cpp](#).

References [mdata::TestParameters< T >::alg](#), [mdata::TestParameters< T >::alpha](#), [ThreadPool::enqueue\(\)](#), [mdata::TestParameters< T >::execTimesCol](#), [mdata::TestParameters< T >::execTimesRow](#), [mdata::TestParameters< T >::execTimesTable](#), [mdata::DataTable< T >::exportCSV\(\)](#), [mdata::TestParameters< T >::funcId](#), [mfunc::FunctionDesc::get\(\)](#), [mfunc::NUM_FUNCTIONS](#), [mdata::TestParameters< T >::resultsCol](#), [mdata::TestParameters< T >::resultsRow](#), [mdata::TestParameters< T >::resultsTable](#), [mdata::DataTable< T >::setColLabel\(\)](#), and [ThreadPool::stopAndJoinAll\(\)](#).

Referenced by [runExp\(\)](#).

```

00192 {
00193     if (populationsPool.size() == 0) return 1;
00194
00195     // Construct results and execution times tables
00196     mdata::DataTable<T> resultsTable(iterations, (size_t)
NUM_FUNCTIONS);
00197     mdata::DataTable<T> execTimesTable(iterations, (size_t)
NUM_FUNCTIONS);
00198
00199     // Prepare thread futures vector, used to ensure all async tasks complete
00200     // successfully.
00201     std::vector<std::future<int>> testFutures;

```

```

00202
00203 // Start recording total execution time
00204 high_resolution_clock::time_point t_start = high_resolution_clock::now();
00205
00206 // For each of the NUM_FUNCTIONS functions, prepare a TestParameters
00207 // struct and queue an asynchronous test that will be picked up and
00208 // executed by one of the threads in the thread pool.
00209 for (unsigned int i = 0; i < NUM_FUNCTIONS; i++)
00210 {
00211     // Update column labels for results and exec times tables
00212     resultsTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
00213     execTimesTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
00214
00215     // Queue up a new function test for each iteration
00216     for (size_t iter = 0; iter < iterations; iter++)
00217     {
00218         mdata::TestParameters<T> curParam;
00219         curParam.funcId = i + 1;
00220         curParam.alpha = alpha;
00221         curParam.alg = testAlg;
00222         curParam.resultsTable = &resultsTable;
00223         curParam.execTimesTable = &execTimesTable;
00224         curParam.resultsCol = i;
00225         curParam.execTimesCol = i;
00226         curParam.resultsRow = iter;
00227         curParam.execTimesRow = iter;
00228
00229         // Add function test to async queue
00230         testFutures.emplace_back(
00231             tPool->enqueue(&Experiment<T>::testFuncThreaded, this
, curParam)
00232         );
00233     }
00234 }
00235
00236 // Get the total number of async tasks queued
00237 const double totalFutures = static_cast<double>(testFutures.size());
00238 int tensPercentile = -1;
00239 std::chrono::microseconds waitTime(100);
00240
00241 // Loop until all async tasks are completed and the thread futures
00242 // array is empty
00243 while (testFutures.size() > 0)
00244 {
00245     // Sleep a little bit since the async thread tasks are higher priority
00246     std::this_thread::sleep_for(waitTime);
00247
00248     // Get iterator to first thread future
00249     auto it = testFutures.begin();
00250
00251     // Loop through all thread futures
00252     while (it != testFutures.end())
00253     {
00254         if (!it->valid())
00255         {
00256             // An error occurred with one of the threads
00257             cerr << "Error: Thread future invalid.";
00258             tPool->stopAndJoinAll();
00259             return 1;
00260         }
00261
00262         // Get the status of the current thread future (async task)
00263         std::future_status status = it->wait_for(waitTime);
00264         if (status == std::future_status::ready)
00265         {
00266             // Task has completed, get return value
00267             int errCode = it->get();
00268             if (errCode)
00269             {
00270                 // An error occurred while running the task.
00271                 // Bail out of function
00272                 tPool->stopAndJoinAll();
00273                 return errCode;
00274             }
00275
00276             // Remove processed task future from vector
00277             it = testFutures.erase(it);
00278
00279             // Calculate the percent completed of all tasks, rounded to the nearest 10%
00280             int curPercentile = static_cast<int>(((totalFutures - testFutures.size()) / totalFutures) *
10);
00281
00282             if (curPercentile > tensPercentile)
00283             {
00284                 // Print latest percent value to the console
00285                 tensPercentile = curPercentile;
00286                 cout << "~" << (tensPercentile * 10) << "% " << flush;

```

```

00287         }
00288         else
00289         {
00290             // Async task has not yet completed, advance to the next one
00291             it++;
00292         }
00293     }
00294 }
00295
00296 // Record total execution time and print it to the console
00297 high_resolution_clock::time_point t_end = high_resolution_clock::now();
00298 long double totalExecTime = static_cast<long double>(duration_cast<nanoseconds>(t_end - t_start).count(
00299 )) / 1000000000.0L;
00300
00301 cout << endl << "Test finished. Total time: " << std::setprecision(7) << totalExecTime << " seconds." <
00302 < endl;
00303
00304 if (!resultsFile.empty())
00305 {
00306     // Export results table to a *.csv file
00307     cout << "Exporting results to: " << resultsFile << endl;
00308     resultsTable.exportCSV(resultsFile.c_str());
00309 }
00310
00311 if (!execTimesFile.empty())
00312 {
00313     // Export exec times table to a *.csv file
00314     cout << "Exporting execution times to: " << execTimesFile << endl;
00315     execTimesTable.exportCSV(execTimesFile.c_str());
00316 }
00317
00318 cout << flush;
00319
00320 return 0;
00321 }

```

6.4.3.3 testFuncThreaded()

```

template<class T >
int Experiment::testFuncThreaded (
    mdata::TestParameters< T > tParams )

```

Executes a single iteration of a test with the given parameters.

Template Parameters

<i>T</i>	The data type used by the test
----------	--------------------------------

Parameters

<i>tParams</i>	The parameters used to set up the test
----------------	--

Returns

int An error code if any

Definition at line 329 of file [experiment.cpp](#).

References [mdata::TestParameters< T >::alg](#), [mdata::TestParameters< T >::alpha](#), [enums::BlindSearch](#), [mdata::TestParameters< T >::execTimesCol](#), [mdata::TestParameters< T >::execTimesRow](#), [mdata::TestParameters< T >::execTimesTable](#), [mdata::TestParameters< T >::funcId](#), [util::IniReader::getEntry\(\)](#), [enums::LocalSearch](#), [mfunc::RandomBounds< T >::max](#), [mfunc::RandomBounds< T >::min](#), [mfunc::NUM_FUNCTIONS](#), [mdata::TestParameters< T >::resultsCol](#), [mdata::TestParameters< T >::resultsRow](#), [mdata::TestParameters< T >::resultsTable](#), and [mdata::SearchAlgorithm< T >::run\(\)](#).

```

00330 {
00331     mdata::SearchAlgorithm<T>* alg;
00332
00333     // Construct a search algorithm object for the selected alg
00334     switch (tParams.alg)
00335     {
00336         case enums::Algorithm::BlindSearch:
00337             alg = new mdata::BlindSearch<T>();
00338             break;
00339         case enums::Algorithm::LocalSearch:
00340             alg = new mdata::LocalSearch<T>();
00341             break;
00342         default:
00343             cerr << "Invalid algorithm selected." << endl;
00344             return 1;
00345     }
00346
00347     // Retrieve the function bounds
00348     const RandomBounds<T>& funcBounds = vBounds[tParams.funcId - 1];
00349
00350     // Retrieve the next available population object from the population pool
00351     mdata::Population<T>* pop = popPoolRemove();
00352
00353     // Run the search algorithm one and record the results
00354     auto tResult = alg->run(Functions<T>::get(tParams.funcId), funcBounds.
min, funcBounds.max, pop, tParams.alpha);
00355
00356     // Place the population object back into the pool to be reused by another thread
00357     popPoolAdd(pop);
00358
00359     if (tResult.err)
00360     {
00361         cerr << "Error while testing function " << tParams.funcId << endl;
00362         return tResult.err;
00363     }
00364
00365     // Update results table and execution times table with algorithm results
00366     tParams.resultsTable->setEntry(tParams.resultsRow, tParams.
resultsCol, tResult.fitness);
00367     tParams.execTimesTable->setEntry(tParams.execTimesRow, tParams.
execTimesCol, tResult.execTime);
00368
00369     delete alg;
00370     return 0;
00371 }

```

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

- include/experiment.h
- src/experiment.cpp

6.5 mfunc::FunctionDesc Struct Reference

[get\(\)](#) returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null

```
#include <mfunctions.h>
```

Static Public Member Functions

- static const char * [get](#) (unsigned int f)

6.5.1 Detailed Description

[get\(\)](#) returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null

Parameters

<i>f</i>	Function id to retrieve the description for
----------	---

Returns

A C-string containing the function description if id is valid, otherwise null.

Definition at line 56 of file [mfunctions.h](#).

6.5.2 Member Function Documentation

6.5.2.1 get()

```
static const char* mfunc::FunctionDesc::get (
    unsigned int f ) [inline], [static]
```

Definition at line 58 of file [mfunctions.h](#).

References [_ackleysOneDesc](#), [_ackleysTwoDesc](#), [_alpineDesc](#), [_dejongDesc](#), [_eggHolderDesc](#), [_griewangk↵Desc](#), [_levyDesc](#), [_mastersCosineWaveDesc](#), [_michalewiczDesc](#), [_pathologicalDesc](#), [_quarticDesc](#), [_ranaDesc](#), [_rastriginDesc](#), [_rosenbrokDesc](#), [_schwefelDesc](#), [_sineEnvelopeSineWaveDesc](#), [_stepDesc](#), and [_stretchedV↵SineWaveDesc](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#).

```
00059     {
00060         switch (f)
00061         {
00062             case 1:
00063                 return _schwefelDesc;
00064             case 2:
00065                 return _dejongDesc;
00066             case 3:
00067                 return _rosenbrokDesc;
00068             case 4:
00069                 return _rastriginDesc;
00070             case 5:
00071                 return _griewangkDesc;
00072             case 6:
00073                 return _sineEnvelopeSineWaveDesc;
00074             case 7:
00075                 return _stretchedVSineWaveDesc;
00076             case 8:
00077                 return _ackleysOneDesc;
00078             case 9:
00079                 return _ackleysTwoDesc;
00080             case 10:
00081                 return _eggHolderDesc;
00082             case 11:
00083                 return _ranaDesc;
00084             case 12:
00085                 return _pathologicalDesc;
00086             case 13:
00087                 return _michalewiczDesc;
00088             case 14:
00089                 return _mastersCosineWaveDesc;
00090             case 15:
00091                 return _quarticDesc;
00092             case 16:
00093                 return _levyDesc;
00094             case 17:
00095                 return _stepDesc;
00096             case 18:
00097                 return _alpineDesc;
00098             default:
00099                 return NULL;
00100         }
00101     }
```

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

- include/[mfunctions.h](#)

6.6 mfunc::Functions< T > Struct Template Reference

Struct containing all static math functions. A function can be called directly by name, or indirectly using [Functions::get](#) or [Functions::exec](#).

```
#include <mfunctions.h>
```

Static Public Member Functions

- static T [schwefel](#) (T *v, size_t n)
Function 1. Implementation of Schwefel's mathematical function.
- static T [dejong](#) (T *v, size_t n)
Function 2. Implementation of 1st De Jong's mathematical function.
- static T [rosenbrok](#) (T *v, size_t n)
Function 3. Implementation of the Rosenbrock mathematical function.
- static T [rastrigin](#) (T *v, size_t n)
Function 4. Implementation of the Rastrigin mathematical function.
- static T [griewangk](#) (T *v, size_t n)
Function 5. Implementation of the Griewangk mathematical function.
- static T [sineEnvelopeSineWave](#) (T *v, size_t n)
Function 6. Implementation of the Sine Envelope Sine Wave mathematical function.
- static T [stretchedVSineWave](#) (T *v, size_t n)
Function 7. Implementation of the Stretched V Sine Wave mathematical function.
- static T [ackleysOne](#) (T *v, size_t n)
Function 8. Implementation of Ackley's One mathematical function.
- static T [ackleysTwo](#) (T *v, size_t n)
Function 9. Implementation of Ackley's Two mathematical function.
- static T [eggHolder](#) (T *v, size_t n)
Function 10. Implementation of the Egg Holder mathematical function.
- static T [rana](#) (T *v, size_t n)
Function 11. Implementation of the Rana mathematical function.
- static T [pathological](#) (T *v, size_t n)
Function 12. Implementation of the Pathological mathematical function.
- static T [mastersCosineWave](#) (T *v, size_t n)
Function 14. Implementation of the Masters Cosine Wave mathematical function.
- static T [michalewicz](#) (T *v, size_t n)
Function 13. Implementation of the Michalewicz mathematical function.
- static T [quartic](#) (T *v, size_t n)
Function 15. Implementation of the Quartic mathematical function.
- static T [levy](#) (T *v, size_t n)
Function 16. Implementation of the Levy mathematical function.
- static T [step](#) (T *v, size_t n)
Function 17. Implementation of the Step mathematical function.
- static T [alpine](#) (T *v, size_t n)

Function 18. Implementation of the Alpine mathematical function.

- static `mfuncPtr< T > get` (unsigned int f)

Returns a function pointer to the math function with the given id.

- static `bool exec` (unsigned int f, T *v, size_t n, T &outResult)

Executes a specific function Executes the function with the given id and returns true on success. Otherwise returns false if id is invalid.

- static `T nthroot` (T x, T n)
- static `T w` (T x)

6.6.1 Detailed Description

```
template<class T>
struct mfunc::Functions< T >
```

Struct containing all static math functions. A function can be called directly by name, or indirectly using [Functions::get](#) or [Functions::exec](#).

Template Parameters

<code>T</code>	Data type for function calculations
----------------	-------------------------------------

Definition at line 112 of file [mfunctions.h](#).

6.6.2 Member Function Documentation

6.6.2.1 `ackleysOne()`

```
template<class T >
T mfunc::Functions< T >::ackleysOne (
    T * v,
    size_t n ) [static]
```

Function 8. Implementation of Ackley's One mathematical function.

Parameters

<code>v</code>	Vector as a T value array
<code>n</code>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 331 of file [mfunctions.h](#).


```

00332 {
00333     T f = 0.0;
00334
00335     for (size_t i = 0; i < n - 1; i++)
00336     {
00337         T a = (static_cast<T>(1.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2))) * sqrt(v[i]*v[i] + v[i+
1]*v[i+1]);
00338         T b = static_cast<T>(3.0) * (cos(static_cast<T>(2.0) * v[i]) + sin(static_cast<T>(2.0) * v[i+1]));
00339         f += a + b;
00340     }
00341
00342     return f;
00343 }

```

6.6.2.2 ackleysTwo()

```

template<class T >
T mfunc::Functions< T >::ackleysTwo (
    T * v,
    size_t n ) [static]

```

Function 9. Implementation of Ackley's Two mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 355 of file [mfunctions.h](#).

```

00356 {
00357     T f = 0.0;
00358
00359     for (size_t i = 0; i < n - 1; i++)
00360     {
00361         T a = static_cast<T>(20.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2) * sqrt((v[i]*v[i] + v[i+1
]*v[i+1]) / static_cast<T>(2.0)));
00362         T b = pow(static_cast<T>(M_E), static_cast<T>(0.5) *
00363             (cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + cos(static_cast<T>(2.0) *
static_cast<T>(M_PI) * v[i+1])));
00364         f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
00365     }
00366
00367     return f;
00368 }

```

6.6.2.3 alpine()

```

template<class T >
T mfunc::Functions< T >::alpine (
    T * v,
    size_t n ) [static]

```

Function 18. Implementation of the Alpine mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 585 of file [mfunctions.h](#).

```

00586 {
00587     T f = 0.0;
00588
00589     for (size_t i = 0; i < n; i++)
00590     {
00591         f += std::abs(v[i] * sin(v[i]) + static_cast<T>(0.1)*v[i]);
00592     }
00593
00594     return f;
00595 }
```

6.6.2.4 dejong()

```

template<class T >
T mfunc::Functions< T >::dejong (
    T * v,
    size_t n ) [static]
```

Function 2. Implementation of 1st De Jong's mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 183 of file [mfunctions.h](#).

```

00184 {
00185     T f = 0.0;
00186
00187     for (size_t i = 0; i < n; i++)
00188     {
00189         f += v[i] * v[i];
00190     }
00191
00192     return f;
00193 }
```

6.6.2.5 eggHolder()

```
template<class T >
T mfunc::Functions< T >::eggHolder (
    T * v,
    size_t n ) [static]
```

Function 10. Implementation of the Egg Holder mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 380 of file [mfunctions.h](#).

```
00381 {
00382     T f = 0.0;
00383
00384     for (size_t i = 0; i < n - 1; i++)
00385     {
00386         T a = static_cast<T>(-1.0) * v[i] * sin(sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0))));
00387         T b = (v[i+1] + static_cast<T>(47)) * sin(sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (v[i]/
00388             static_cast<T>(2.0))));
00389         f += a - b;
00390     }
00391     return f;
00392 }
```

6.6.2.6 exec()

```
template<class T >
bool mfunc::Functions< T >::exec (
    unsigned int f,
    T * v,
    size_t n,
    T & outResult ) [static]
```

Executes a specific function Executes the function with the given id and returns true on success. Otherwise returns false if id is invalid.

Parameters

<i>f</i>	Function id to execute
<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'
<i>outResult</i>	Output reference variable for the result of the mathematical function

Returns

true if 'f' is a valid id and the function was ran. Otherwise false.

Definition at line 667 of file [mfunctions.h](#).

```
00668 {
00669     auto fPtr = get(f);
00670     if (fPtr == nullptr) return false;
00671
00672     outResult = fPtr(v, n);
00673     return true;
00674 }
```

6.6.2.7 get()

```
template<class T >
mfunc::mfuncPtr< T > mfunc::Functions< T >::get (
    unsigned int f ) [static]
```

Returns a function pointer to the math function with the given id.

Template Parameters

<i>T</i>	Data type to be used in the function's calculations
----------	---

Parameters

<i>f</i>	Id of the function (1-18)
----------	---------------------------

Returns

[mfunc::mfuncPtr<T>](#) Function pointer to the associated function, or nullptr if the id is invalid.

Definition at line 609 of file [mfunctions.h](#).

```
00610 {
00611     switch (f)
00612     {
00613         case 1:
00614             return Functions<T>::schwefel;
00615         case 2:
00616             return Functions<T>::dejong;
00617         case 3:
00618             return Functions<T>::rosenbrok;
00619         case 4:
00620             return Functions<T>::rastrigin;
00621         case 5:
00622             return Functions<T>::griewangk;
00623         case 6:
00624             return Functions<T>::sineEnvelopeSineWave;
00625         case 7:
00626             return Functions<T>::stretchedVSineWave;
00627         case 8:
00628             return Functions<T>::ackleysOne;
00629         case 9:
00630             return Functions<T>::ackleysTwo;
00631         case 10:
```

```

00632         return Functions<T>::eggHolder;
00633     case 11:
00634         return Functions<T>::rana;
00635     case 12:
00636         return Functions<T>::pathological;
00637     case 13:
00638         return Functions<T>::michalewicz;
00639     case 14:
00640         return Functions<T>::mastersCosineWave;
00641     case 15:
00642         return Functions<T>::quartic;
00643     case 16:
00644         return Functions<T>::levy;
00645     case 17:
00646         return Functions<T>::step;
00647     case 18:
00648         return Functions<T>::alpine;
00649     default:
00650         return nullptr;
00651     }
00652 }

```

6.6.2.8 griewangk()

```

template<class T >
T mfunc::Functions< T >::griewangk (
    T * v,
    size_t n ) [static]

```

Function 5. Implementation of the Griewangk mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 252 of file [mfunctions.h](#).

```

00253 {
00254     T sum = 0.0;
00255     T product = 0.0;
00256
00257     for (size_t i = 0; i < n; i++)
00258     {
00259         sum += (v[i] * v[i]) / static_cast<T>(4000.0);
00260     }
00261
00262     for (size_t i = 0; i < n; i++)
00263     {
00264         product *= cos(v[i] / sqrt(static_cast<T>(i + 1.0)));
00265     }
00266
00267     return static_cast<T>(1.0) + sum - product;
00268 }

```

6.6.2.9 levy()

```
template<class T >
T mfunc::Functions< T >::levy (
    T * v,
    size_t n ) [static]
```

Function 16. Implementation of the Levy mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 531 of file [mfunctions.h](#).

```
00532 {
00533     T f = 0.0;
00534
00535     for (size_t i = 0; i < n - 1; i++)
00536     {
00537         T a = w(v[i]) - static_cast<T>(1.0);
00538         a *= a;
00539         T b = sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
00540         b *= b;
00541         T c = w(v[n - 1]) - static_cast<T>(1.0);
00542         c *= c;
00543         T d = sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
00544         d *= d;
00545         f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
00546     }
00547
00548     T e = sin(static_cast<T>(M_PI) * w(v[0]));
00549     return e*e + f;
00550 }
```

6.6.2.10 mastersCosineWave()

```
template<class T >
T mfunc::Functions< T >::mastersCosineWave (
    T * v,
    size_t n ) [static]
```

Function 14. Implementation of the Masters Cosine Wave mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 476 of file [mfunctions.h](#).

```
00477 {
00478     T f = 0.0;
00479     for (size_t i = 0; i < n - 1; i++)
00480     {
00481         T a = pow(M_E, static_cast<T>(-1.0/8.0)*(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i+1]*v[i
00482     ]));
00483         T b = cos(static_cast<T>(4) * sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v[i+1]));
00484         f += a * b;
00485     }
00486     return static_cast<T>(-1.0) * f;
00487 }
00488 }
```

6.6.2.11 michalewicz()

```
template<class T >
T mfunc::Functions< T >::michalewicz (
    T * v,
    size_t n ) [static]
```

Function 13. Implementation of the Michalewicz mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 454 of file [mfunctions.h](#).

```
00455 {
00456     T f = 0.0;
00457     for (size_t i = 0; i < n; i++)
00458     {
00459         f += sin(v[i]) * pow(sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)), static_cast<T>(20));
00460     }
00461     return -1.0 * f;
00462 }
00463 }
00464 }
```

6.6.2.12 nthroot()

```
template<class T >
T mfunc::Functions< T >::nthroot (
    T x,
    T n ) [static]
```

Simple helper function that returns the nth-root

Parameters

x	Value to be taken to the nth power
n	root degree

Returns

The value of the nth-root of x

Definition at line 146 of file [mfunctions.h](#).

```
00147 {
00148     return pow(x, static_cast<T>(1.0) / n);
00149 }
```

6.6.2.13 pathological()

```
template<class T >
T mfunc::Functions< T >::pathological (
    T * v,
    size_t n ) [static]
```

Function 12. Implementation of the Pathological mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 428 of file [mfunctions.h](#).

```
00429 {
00430     T f = 0.0;
00431
00432     for (size_t i = 0; i < n - 1; i++)
00433     {
00434         T a = sin(sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
00435         a = (a*a) - static_cast<T>(0.5);
00436         T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
00437         b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
00438         f += static_cast<T>(0.5) + (a/b);
00439     }
00440
00441     return f;
00442 }
```

6.6.2.14 quartic()

```
template<class T >
T mfunc::Functions< T >::quartic (
    T * v,
    size_t n ) [static]
```

Function 15. Implementation of the Quartic mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 500 of file [mfunctions.h](#).

```
00501 {
00502     T f = 0.0;
00503
00504     for (size_t i = 0; i < n; i++)
00505     {
00506         f += (i+1) * v[i] * v[i] * v[i] * v[i];
00507     }
00508
00509     return f;
00510 }
```

6.6.2.15 rana()

```
template<class T >
T mfunc::Functions< T >::rana (
    T * v,
    size_t n ) [static]
```

Function 11. Implementation of the Rana mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 404 of file [mfunctions.h](#).

```

00405 {
00406     T f = 0.0;
00407
00408     for (size_t i = 0; i < n - 1; i++)
00409     {
00410         T a = v[i] * sin(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * cos(sqrt(std::abs(v[i+1] +
v[i] + static_cast<T>(1.0))));
00411         T b = (v[i+1] + static_cast<T>(1.0)) * cos(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) *
sin(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00412         f += a + b;
00413     }
00414
00415     return f;
00416 }

```

6.6.2.16 rastrigin()

```

template<class T >
T mfunc::Functions< T >::rastrigin (
    T * v,
    size_t n ) [static]

```

Function 4. Implementation of the Rastrigin mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 230 of file [mfunctions.h](#).

```

00231 {
00232     T f = 0.0;
00233
00234     for (size_t i = 0; i < n; i++)
00235     {
00236         f += (v[i] * v[i]) - (static_cast<T>(10.0) * cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]))
    };
00237 }
00238
00239 return static_cast<T>(10.0) * static_cast<T>(n) * f;
00240 }

```

6.6.2.17 rosenbrok()

```

template<class T >
T mfunc::Functions< T >::rosenbrok (
    T * v,
    size_t n ) [static]

```

Function 3. Implementation of the Rosenbrock mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 205 of file [mfunctions.h](#).

```

00206 {
00207     T f = 0.0;
00208
00209     for (size_t i = 0; i < n - 1; i++)
00210     {
00211         T a = ((v[i] * v[i]) - v[i+1]);
00212         T b = (static_cast<T>(1.0) - v[i]);
00213         f += static_cast<T>(100.0) * a * a;
00214         f += b * b;
00215     }
00216
00217     return f;
00218 }
```

6.6.2.18 schwefel()

```

template<class T >
T mfunc::Functions< T >::schwefel (
    T * v,
    size_t n ) [static]
```

Function 1. Implementation of Schwefel's mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 161 of file [mfunctions.h](#).

```

00162 {
00163     T f = 0.0;
00164
00165     for (size_t i = 0; i < n; i++)
00166     {
00167         f += (static_cast<T>(-1.0) * v[i]) * sin(sqrt(std::abs(v[i])));
00168     }
00169
00170     return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
00171 }
```

6.6.2.19 sineEnvelopeSineWave()

```
template<class T >
T mfunc::Functions< T >::sineEnvelopeSineWave (
    T * v,
    size_t n ) [static]
```

Function 6. Implementation of the Sine Envelope Sine Wave mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 280 of file [mfunctions.h](#).

```
00281 {
00282     T f = 0.0;
00283
00284     for (size_t i = 0; i < n - 1; i++)
00285     {
00286         T a = sin(v[i]*v[i] + v[i+1]*v[i+1] - static_cast<T>(0.5));
00287         a *= a;
00288         T b = (static_cast<T>(1.0) + static_cast<T>(0.001)*(v[i]*v[i] + v[i+1]*v[i+1]));
00289         b *= b;
00290         f += static_cast<T>(0.5) + (a / b);
00291     }
00292
00293     return static_cast<T>(-1.0) * f;
00294 }
```

6.6.2.20 step()

```
template<class T >
T mfunc::Functions< T >::step (
    T * v,
    size_t n ) [static]
```

Function 17. Implementation of the Step mathematical function.

Parameters

<i>v</i>	Vector as a T value array
<i>n</i>	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 562 of file [mfunctions.h](#).

```

00563 {
00564     T f = 0.0;
00565
00566     for (size_t i = 0; i < n; i++)
00567     {
00568         T a = std::abs(v[i]) + static_cast<T>(0.5);
00569         f += a * a;
00570     }
00571
00572     return f;
00573 }
```

6.6.2.21 stretchedVSineWave()

```

template<class T >
T mfunc::Functions< T >::stretchedVSineWave (
    T * v,
    size_t n ) [static]
```

Function 7. Implementation of the Stretched V Sine Wave mathematical function.

Parameters

v	Vector as a T value array
n	Size of the vector 'v'

Returns

The result of the mathematical function

Definition at line 306 of file [mfunctions.h](#).

```

00307 {
00308     T f = 0.0;
00309
00310     for (size_t i = 0; i < n - 1; i++)
00311     {
00312         T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
00313         T b = sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0)));
00314         b *= b;
00315         f += a * b + static_cast<T>(1.0);
00316     }
00317
00318     return f;
00319 }
```

6.6.2.22 w()

```

template<class T >
T mfunc::Functions< T >::w (
    T x ) [static]
```

Helper math function used in [levy\(\)](#)

Definition at line 518 of file [mfunctions.h](#).

```

00519 {
00520     return static_cast<T>(1.0) + (x - static_cast<T>(1.0)) / static_cast<T>(4.0);
00521 }

```

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

- include/mfunctions.h

6.7 util::IniReader Class Reference

The [IniReader](#) class is a simple *.ini file reader and parser.

```
#include <inireader.h>
```

Public Member Functions

- [IniReader](#) ()
Construct a new [IniReader](#) object.
- [~IniReader](#) ()
Destroys the [IniReader](#) object.
- bool [openFile](#) (std::string filePath)
Opens the given ini file and parses all sections/entries. The all file data is stored in memory and the file is closed.
- bool [sectionExists](#) (std::string section)
Returns true if the given section exists in the current ini file.
- bool [entryExists](#) (std::string section, std::string entry)
Returns true if the given section and entry key exists in the current ini file.
- std::string [getEntry](#) (std::string section, std::string entry)
Returns the value for the entry that has the given entry key within the given section.
- template<class T >
T [getEntryAs](#) (std::string section, std::string entry)

6.7.1 Detailed Description

The [IniReader](#) class is a simple *.ini file reader and parser.

– Initialize an [IniReader](#) object:

```
IniReader ini;
```

Open and parse an *.ini file:

```
ini.openFile("my_ini_file.ini");
```

Note that the file is immediately closed after parsing, and the file data is retained in memory.

Retrieve an entry from the ini file:

```
std::string value = ini.getEntry("My Section", "entryKey");
```

Definition at line 46 of file [inireader.h](#).

6.7.2 Constructor & Destructor Documentation

6.7.2.1 IniReader()

```
IniReader::IniReader ( )
```

Construct a new [IniReader](#) object.

Definition at line 21 of file [inireader.cpp](#).

```
00021             : file(""), iniMap()
00022 {
00023 }
```

6.7.2.2 ~IniReader()

```
IniReader::~~IniReader ( )
```

Destroys the [IniReader](#) object.

Definition at line 28 of file [inireader.cpp](#).

```
00029 {
00030     iniMap.clear();
00031 }
```

6.7.3 Member Function Documentation

6.7.3.1 entryExists()

```
bool IniReader::entryExists (
    std::string section,
    std::string entry )
```

Returns true if the given section and entry key exists in the current ini file.

Parameters

<i>section</i>	std::string containing the section name
<i>entry</i>	std::string containing the entry key name

Returns

Returns true if the section and entry key exist in the ini file, otherwise false.

Definition at line 67 of file [inireader.cpp](#).

Referenced by [getEntry\(\)](#).

```
00068 {
00069     auto it = iniMap.find(section);
00070     if (it == iniMap.end()) return false;
00071
00072     return it->second.find(entry) != it->second.end();
00073 }
```

6.7.3.2 getEntry()

```
std::string IniReader::getEntry (
    std::string section,
    std::string entry )
```

Returns the value for the entry that has the given entry key within the given section.

Parameters

<i>section</i>	std::string containing the section name
<i>entry</i>	std::string containing the entry key name

Returns

The value of the entry with the given entry key and section. Returns an empty string if the entry does not exist.

Definition at line 84 of file [inireader.cpp](#).

References [entryExists\(\)](#).

Referenced by [getEntryAs\(\)](#), [mfunc::Experiment< T >::init\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

```
00085 {
00086     if (!entryExists(section, entry)) return std::string();
00087
00088     return iniMap[section][entry];
00089 }
```

6.7.3.3 getEntryAs()

```
template<class T >
T util::IniReader::getEntryAs (
    std::string section,
    std::string entry ) [inline]
```

Definition at line 57 of file [inireader.h](#).

References [getEntry\(\)](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

```
00058     {
00059         std::stringstream ss(getEntry(section, entry));
00060         T retVal;
00061         ss >> retVal;
00062         return retVal;
00063     }
```

6.7.3.4 openFile()

```
bool IniReader::openFile (
    std::string filePath )
```

Opens the given ini file and parses all sections/entries. The all file data is stored in memory and the file is closed.

Parameters

<i>filePath</i>	Path to the ini file you wish to open
-----------------	---------------------------------------

Returns

Returns true if the file was succesfully opened and parsed. Otherwise false.

Definition at line 40 of file [inireader.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

```
00041 {
00042     file = filePath;
00043     if (!parseFile())
00044         return false;
00045
00046     return true;
00047 }
```

6.7.3.5 sectionExists()

```
bool IniReader::sectionExists (
    std::string section )
```

Returns true if the given section exists in the current ini file.

Parameters

<i>section</i>	std::string containing the section name
----------------	---

Returns

Returns true if the section exists in the ini file, otherwise false.

Definition at line 55 of file [inireader.cpp](#).

```
00056 {
00057     return iniMap.find(section) != iniMap.end();
00058 }
```

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

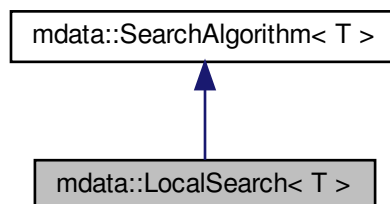
- include/[inireader.h](#)
- src/[inireader.cpp](#)

6.8 mdata::LocalSearch< T > Class Template Reference

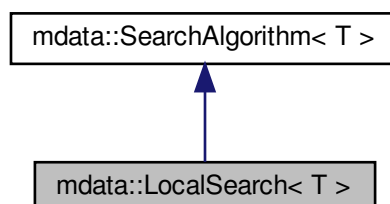
The [LocalSearch](#) class implements the Local Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.

```
#include <localsearch.h>
```

Inheritance diagram for mdata::LocalSearch< T >:



Collaboration diagram for mdata::LocalSearch< T >:



Public Member Functions

- virtual [TestResult](#)< T > [run](#) ([mfunc::mfuncPtr](#)< T > *funcPtr*, const T *fMin*, const T *fMax*, [Population](#)< T > *const *pop*, const T *alpha*)

Executes Local Search with the given population and parameters.

Additional Inherited Members

6.8.1 Detailed Description

```
template<class T>
class mdata::LocalSearch< T >
```

The [LocalSearch](#) class implements the Local Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.

Template Parameters

<i>T</i>	Data type used
----------	----------------

Definition at line 32 of file [localsearch.h](#).

6.8.2 Member Function Documentation

6.8.2.1 run()

```
template<class T >
virtual TestResult<T> mdata::LocalSearch< T >::run (
    mfunc::mfuncPtr< T > funcPtr,
    const T fMin,
    const T fMax,
    Population< T > *const pop,
    const T alpha ) [inline], [virtual]
```

Executes Local Search with the given population and parameters.

Parameters

<i>funcPtr</i>	Function pointer to the math function being used to generate the population
<i>fMin</i>	Minimum bound for the population matrix vector components
<i>fMax</i>	Maximum bound for the population matrix vector components
<i>pop</i>	Pointer to a population object that will be used in the local search
<i>alpha</i>	Alpha value for local search neighbor generation

Returns

TestResult<T> Returns a [TestResult](#) struct containing the error code, fitness, and execution time

Implements [mdata::SearchAlgorithm< T >](#).

Definition at line 49 of file [localsearch.h](#).

References [mdata::Population< T >::calcFitness\(\)](#), [mdata::Population< T >::generate\(\)](#), [mdata::Population< T >::getBestFitnessIndex\(\)](#), [mdata::Population< T >::getDimensionsSize\(\)](#), [mdata::Population< T >::getFitness\(\)](#), [mdata::Population< T >::getPopulationPtr\(\)](#), [mdata::Population< T >::getPopulationSize\(\)](#), [mdata::SearchAlgorithm< T >::startTimer\(\)](#), and [mdata::SearchAlgorithm< T >::stopTimer\(\)](#).

```

00050     {
00051         // Get population size and dimensions
00052         const size_t popSize = pop->getPopulationSize();
00053         const size_t dimSize = pop->getDimensionsSize();
00054
00055         // Make sure funcPtr is valid;
00056         if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with
error code 1
00057
00058         // Algorithm related variables
00059         bool stop = false;
00060         size_t pIndex = 0;
00061
00062         startTimer();
00063
00064         // Start recording execution time
00065         pop->generate(fMin, fMax);
00066
00067         for (size_t sol = 0; sol < popSize; sol++)
00068         {
00069             // Populate fitness values using given math function pointer
00070             if (!pop->calcFitness(sol, funcPtr))
00071                 return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
00072         }
00073
00074         // Get the index for the best fitness in the population
00075         pIndex = pop->getBestFitnessIndex();
00076
00077         // Get population vector and fitness of best solution
00078         T* x = pop->getPopulationPtr(pIndex);
00079         T xFit = pop->getFitness(pIndex);
00080
00081         // Create empty Y vector
00082         T* y = util::allocArray<T>(dimSize);
00083         T yFit = 0;
00084
00085         // Create empty Z vector
00086         T* z = util::allocArray<T>(dimSize);
00087         T zFit = 0;
00088
00089         if (x == nullptr || y == nullptr || z == nullptr)
00090         {
00091             std::cerr << "Error in Local Search: Memory allocation failed" << std::endl;
00092             return TestResult<T>(3, 0, 0.0);
00093         }
00094
00095         // Keep looping until search fails to improve
00096         while (!stop)
00097         {
00098             stop = true;
00099
00100             // Copy values from X vector into Y vector
00101             util::copyArray<T>(x, y, dimSize);
00102
00103             // Loop through each dimension in vector
00104             for (size_t a = 0; a < dimSize; a++)
00105             {
00106                 // Add alpha value to y[a]
00107                 y[a] = x[a] + alpha;
00108
00109                 // Make sure y[a] is within the function bounds
00110                 lockBounds(y[a], fMin, fMax);
00111
00112                 // Calculate fitness for y vector
00113                 yFit = funcPtr(y, dimSize);
00114
00115                 // Update Z[a] vector value based on the difference between Y and X

```

```

00116             z[a] = x[a] - (alpha * (yFit - xFit));
00117
00118             // Make sure z[a] is within the function bounds
00119             lockBounds(z[a], fMin, fMax);
00120
00121             y[a] = x[a]; // Reset y[a] to prepare for next loop
00122         }
00123
00124         zFit = funcPtr(z, dimSize);
00125
00126         // The following 'if' statement may cause extreme execution
00127         // times for some functions due to floating point precision:
00128         // if (zFit < xFit)
00129         //
00130         // The replacement 'if' statement below places a limit
00131         // on the minimum acknowledged improvement fitness,
00132         // hopefully preventing extreme run-times:
00133         if (xFit - zFit > MIN_IMPROVEMENT)
00134         {
00135             // Z is an improvement on X, so keep searching
00136             stop = false;
00137
00138             // Swap Z and X for next loop
00139             T* tmp = x;
00140             x = z;
00141             xFit = zFit;
00142             z = tmp;
00143         }
00144     }
00145
00146     // Return best result
00147     return TestResult<T>(0, xFit, stopTimer());
00148 }

```

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

- [include/localsearch.h](#)

6.9 mdata::Population< T > Class Template Reference

Data class for storing a multi-dimensional population of data with the associated fitness.

```
#include <population.h>
```

Public Member Functions

- [Population](#) (size_t popSize, size_t dimensions)
Construct a new [Population](#) object.
- [~Population](#) ()
Destroy [Population](#) object.
- bool [isReady](#) ()
Returns true if the population instance is allocated and ready to be used.
- size_t [getPopulationSize](#) ()
Returns the size of the population.
- size_t [getDimensionsSize](#) ()
Returns the dimensions of the population.
- T * [getPopulationPtr](#) (size_t popIndex)
Returns an array for the population with the given index.
- bool [generate](#) (T minBound, T maxBound)
Generates new random values for this population that are within the given bounds. Resets all fitness values to zero.
- bool [setFitness](#) (size_t popIndex, T value)

- Sets the fitness value for a specific population vector index.*
- bool [calcFitness](#) (size_t popIndex, mfunc::mfuncPtr< T > funcPtr)
Uses the given function pointer to update the fitness value for the population vector at the given index.
- T [getFitness](#) (size_t popIndex)
Returns the fitness value for a specific population vector index.
- T * [getFitnessPtr](#) (size_t popIndex)
Returns the fitness value for a specific population vector index.
- std::vector< T > [getAllFitness](#) ()
Returns a std::vector of all current fitness values.
- T * [getBestFitnessPtr](#) ()
Returns a pointer to the current best fitness value.
- size_t [getBestFitnessIndex](#) ()
Returns the index of the current best fitness value.
- void [outputPopulation](#) (std::ostream &outStream, const char *delim, const char *lineBreak)
Outputs all population data to the given output stream.
- void [outputFitness](#) (std::ostream &outStream, const char *delim, const char *lineBreak)
Outputs all fitness data to the given output stream.

6.9.1 Detailed Description

```
template<class T>
class mdata::Population< T >
```

Data class for storing a multi-dimensional population of data with the associated fitness.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Definition at line 30 of file [population.h](#).

6.9.2 Constructor & Destructor Documentation

6.9.2.1 Population()

```
template<class T >
Population::Population (
    size_t pSize,
    size_t dimensions )
```

Construct a new [Population](#) object.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>pSize</i>	Size of the population.
<i>dimensions</i>	Dimensions of the population.

Definition at line 27 of file [population.cpp](#).

```

00027                                     : popMatrix(nullptr), popSize(pSize), popDim(
      dimensions)
00028 {
00029     if (!allocPopMatrix() || !allocPopFitness())
00030         throw std::bad_alloc();
00031 }
```

6.9.2.2 ~Population()

```

template<class T >
Population::~Population ( )
```

Destroy [Population](#) object.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Definition at line 39 of file [population.cpp](#).

```

00040 {
00041     releasePopMatrix();
00042     releasePopFitness();
00043 }
```

6.9.3 Member Function Documentation**6.9.3.1 calcFitness()**

```

template<class T >
bool Population::calcFitness (
    size_t popIndex,
    mfunc::mfuncPtr< T > funcPtr )
```

Uses the given function pointer to update the fitness value for the population vector at the given index.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>popIndex</i>	Index of the population vector you wish to set the fitness for.
<i>funcPtr</i>	Function pointer to the math function that will be used to calculate the fitness value.

Returns

Returns true on success, otherwise false.

Definition at line 163 of file [population.cpp](#).

Referenced by [mdata::BlindSearch< T >::run\(\)](#), and [mdata::LocalSearch< T >::run\(\)](#).

```

00164 {
00165     if (popFitness == nullptr || popIndex >= popSize) return false;
00166     popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);
00167     return true;
00169 }
00170 }
```

6.9.3.2 generate()

```

template<class T >
bool Population::generate (
    T minBound,
    T maxBound )
```

Generates new random values for this population that are within the given bounds. Resets all fitness values to zero.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>minBound</i>	The minimum bound for a population value.
<i>maxBound</i>	The maximum bound for a population value.

Returns

Returns true if the population was successfully generated, otherwise false.

Definition at line 108 of file [population.cpp](#).

Referenced by [mdata::BlindSearch< T >::run\(\)](#), and [mdata::LocalSearch< T >::run\(\)](#).

```

00109 {
00110     if (popMatrix == nullptr) return false;
```

```

00111
00112     // Generate a new seed for the mersenne twister engine
00113     rgen = std::mt19937(rdev());
00114
00115     // Set up a normal (bell-shaped) distribution for the random number generator with the correct function
    bounds
00116     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00117
00118     // Generate values for all vectors in popMatrix
00119     for (size_t s = 0; s < popSize; s++)
00120     {
00121         for (size_t d = 0; d < popDim; d++)
00122         {
00123             T rand = (T)dist(rgen);
00124             popMatrix[s][d] = rand;
00125         }
00126     }
00127
00128     // Reset popFitness values to 0
00129     initArray<T>(popFitness, popSize, (T)0.0);
00130
00131     return true;
00132 }

```

6.9.3.3 getAllFitness()

```

template<class T >
std::vector< T > Population::getAllFitness ( )

```

Returns a `std::vector` of all current fitness values.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

`std::vector<T>` `std::vector` of fitness values

Definition at line 209 of file [population.cpp](#).

```

00210 {
00211     return std::vector<T>(popFitness[0], popFitness[popSize]);
00212 }

```

6.9.3.4 getBestFitnessIndex()

```

template<class T >
size_t Population::getBestFitnessIndex ( )

```

Returns the index of the current best fitness value.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

size_t Index of the best fitness value

Definition at line 233 of file [population.cpp](#).

Referenced by [mdata::Population< T >::getBestFitnessPtr\(\)](#), and [mdata::LocalSearch< T >::run\(\)](#).

```

00234 {
00235     size_t bestIndex = 0;
00236
00237     for (size_t i = 1; i < popSize; i++)
00238     {
00239         if (popFitness[i] < popFitness[bestIndex])
00240             bestIndex = i;
00241     }
00242
00243     return bestIndex;
00244 }
```

6.9.3.5 getBestFitnessPtr()

```

template<class T >
T * Population::getBestFitnessPtr ( )
```

Returns a pointer to the current best fitness value.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

T* Pointer to the best fitness value

Definition at line 221 of file [population.cpp](#).

References [mdata::Population< T >::getBestFitnessIndex\(\)](#).

Referenced by [mdata::BlindSearch< T >::run\(\)](#).

```

00222 {
00223     return &popFitness[getBestFitnessIndex()];
00224 }
```

6.9.3.6 getDimensionsSize()

```
template<class T >  
size_t Population::getDimensionsSize ( )
```

Returns the dimensions of the population.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

The number of dimensions in the population.

Definition at line 77 of file [population.cpp](#).

Referenced by [mdata::BlindSearch< T >::run\(\)](#), and [mdata::LocalSearch< T >::run\(\)](#).

```
00078 {
00079     return popDim;
00080 }
```

6.9.3.7 getFitness()

```
template<class T >
T Population::getFitness (
    size_t popIndex )
```

Returns the fitness value for a specific population vector index.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>popIndex</i>	Index of the population vector you wish to retrieve the fitness from.
-----------------	---

Returns

Returns the fitness value if popIndex is valid. Otherwise zero.

Definition at line 180 of file [population.cpp](#).

Referenced by [mdata::LocalSearch< T >::run\(\)](#).

```
00181 {
00182     if (popFitness == nullptr || popIndex >= popSize) return 0;
00183     return popFitness[popIndex];
00184 }
00185 }
```

6.9.3.8 getFitnessPtr()

```
template<class T >
T * Population::getFitnessPtr (
    size_t popIndex )
```

Returns the fitness value for a specific population vector index.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>popIndex</i>	Index of the population vector you wish to retrieve the fitness from.
-----------------	---

Returns

Returns the fitness value if popIndex is valid. Otherwise zero.

Definition at line 195 of file [population.cpp](#).

```
00196 {
00197     if (popFitness == nullptr || popIndex >= popSize) return 0;
00198
00199     return &popFitness[popIndex];
00200 }
```

6.9.3.9 getPopulationPtr()

```
template<class T >
T * Population::getPopulationPtr (
    size_t popIndex )
```

Returns an array for the population with the given index.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>popIndex</i>	Index of the population vector you wish to retrieve.
-----------------	--

Returns

Pointer to population vector array at the given index.

Definition at line 90 of file [population.cpp](#).

Referenced by [mdata::LocalSearch< T >::run\(\)](#).

```
00091 {
00092     if (popFitness == nullptr || popIndex >= popSize) return nullptr;
00093
00094     return popMatrix[popIndex];
00095 }
```

6.9.3.10 getPopulationSize()

```
template<class T >
size_t Population::getPopulationSize ( )
```

Returns the size of the population.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

The size of the population.

Definition at line 65 of file [population.cpp](#).

Referenced by [mdata::BlindSearch< T >::run\(\)](#), and [mdata::LocalSearch< T >::run\(\)](#).

```
00066 {
00067     return popSize;
00068 }
```

6.9.3.11 isReady()

```
template<class T >
bool Population::isReady ( )
```

Returns true if the population instance is allocated and ready to be used.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Returns

Returns true if the population instance is in a valid state.

Definition at line 53 of file [population.cpp](#).

```
00054 {
00055     return popMatrix != nullptr && popFitness != nullptr;
00056 }
```

6.9.3.12 outputFitness()

```
template<class T >
void Population::outputFitness (
    std::ostream & outStream,
    const char * delim,
    const char * lineBreak )
```

Outputs all fitness data to the given output stream.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>outStream</i>	Output stream to write the data to.
<i>delim</i>	Delimiter characters to separate columns.
<i>lineBreak</i>	Delimiter characters to separate rows.

Definition at line 281 of file [population.cpp](#).

```
00282 {
00283     if (popFitness == nullptr) return;
00284
00285     for (size_t j = 0; j < popSize; j++)
00286     {
00287         outStream << popFitness[j];
00288         if (j < popSize - 1)
00289             outStream << delim;
00290     }
00291
00292     if (lineBreak != nullptr)
00293         outStream << lineBreak;
00294 }
```

6.9.3.13 outputPopulation()

```
template<class T >
void Population::outputPopulation (
```



```
std::ostream & outStream,
const char * delim,
const char * lineBreak )
```

Outputs all population data to the given output stream.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>outStream</i>	Output stream to write the data to.
<i>delim</i>	Delimiter characters to separate columns.
<i>lineBreak</i>	Delimiter characters to separate rows.

Definition at line 255 of file [population.cpp](#).

```
00256 {
00257     if (popMatrix == nullptr) return;
00258     for (size_t j = 0; j < popSize; j++)
00259     {
00260         for (size_t k = 0; k < popDim; k++)
00261         {
00262             outStream << popMatrix[j][k];
00263             if (k < popDim - 1)
00264                 outStream << delim;
00265         }
00266         outStream << lineBreak;
00267     }
00268 }
00269 }
00270 }
```

6.9.3.14 setFitness()

```
template<class T >
bool Population::setFitness (
    size_t popIndex,
    T value )
```

Sets the fitness value for a specific population vector index.

Template Parameters

<i>T</i>	Data type of the population.
----------	------------------------------

Parameters

<i>popIndex</i>	Index of the population vector you wish to set the fitness for.
<i>value</i>	The value of the fitness.

Returns

Returns true if the fitness was successfully set, otherwise false.

Definition at line 143 of file [population.cpp](#).

```
00144 {
00145     if (popFitness == nullptr || popIndex >= popSize) return false;
00146
00147     popFitness[popIndex] = value;
00148
00149     return true;
00150 }
```

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

- [include/population.h](#)
- [src/population.cpp](#)

6.10 mfunc::RandomBounds< T > Struct Template Reference

Simple struct for storing the minimum and maximum input vector bounds for a function.

```
#include <experiment.h>
```

Public Attributes

- `T min` = 0.0
- `T max` = 0.0

6.10.1 Detailed Description

```
template<class T>
struct mfunc::RandomBounds< T >
```

Simple struct for storing the minimum and maximum input vector bounds for a function.

Definition at line 35 of file [experiment.h](#).

6.10.2 Member Data Documentation

6.10.2.1 max

```
template<class T>
T mfunc::RandomBounds< T >::max = 0.0
```

Definition at line 38 of file [experiment.h](#).

Referenced by [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.10.2.2 min

```
template<class T>
T mfunc::RandomBounds< T >::min = 0.0
```

Definition at line 37 of file [experiment.h](#).

Referenced by [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

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

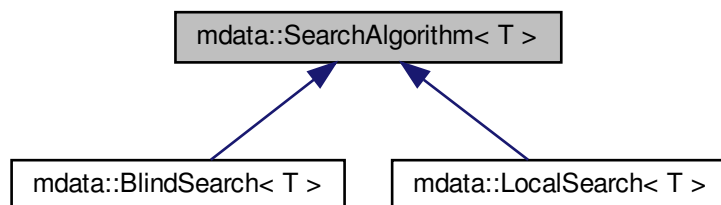
- include/[experiment.h](#)

6.11 mdata::SearchAlgorithm< T > Class Template Reference

The [SearchAlgorithm](#) class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm.

```
#include <searchalg.h>
```

Inheritance diagram for mdata::SearchAlgorithm< T >:



Public Member Functions

- [SearchAlgorithm](#) ()
- virtual `~SearchAlgorithm` ()=0
- virtual `TestResult< T > run` ([mfunc::mfuncPtr< T >](#) funcPtr, const T fMin, const T fMax, [Population< T >](#) *const pop, const T alpha)=0

Protected Member Functions

- void [startTimer](#) ()
Starts the execution time timer.
- double [stopTimer](#) ()
Returns the amount of time that has passed since [startTimer\(\)](#) was called in milliseconds.

Protected Attributes

- double [timeDiff](#)
- [high_resolution_clock::time_point](#) [timer](#)

6.11.1 Detailed Description

```
template<class T>
class mdata::SearchAlgorithm< T >
```

The [SearchAlgorithm](#) class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm.

Template Parameters

T	The data type used by the algorithm
-------------------	-------------------------------------

Definition at line [70](#) of file [searchalg.h](#).

6.11.2 Constructor & Destructor Documentation

6.11.2.1 SearchAlgorithm()

```
template<class T>
mdata::SearchAlgorithm< T >::SearchAlgorithm ( ) [inline]
```

Definition at line [73](#) of file [searchalg.h](#).

```
00073 :   timeDiff(0.0) {}
```

6.11.2.2 ~SearchAlgorithm()

```
template<class T>
mdata::SearchAlgorithm< T >::~~SearchAlgorithm ( ) [pure virtual]
```

Definition at line [102](#) of file [searchalg.h](#).

```
00102 { }
```

6.11.3 Member Function Documentation

6.11.3.1 run()

```
template<class T>
virtual TestResult<T> mdata::SearchAlgorithm< T >::run (
    mfunc::mfuncPtr< T > funcPtr,
    const T fMin,
    const T fMax,
    Population< T > *const pop,
    const T alpha ) [pure virtual]
```

Implemented in [mdata::LocalSearch](#)< T >, and [mdata::BlindSearch](#)< T >.

Referenced by [mfunc::Experiment](#)< T >::testFuncThreaded().

6.11.3.2 startTimer()

```
template<class T>
void mdata::SearchAlgorithm< T >::startTimer ( ) [inline], [protected]
```

Starts the execution time timer.

Definition at line 83 of file [searchalg.h](#).

Referenced by [mdata::BlindSearch](#)< T >::run(), and [mdata::LocalSearch](#)< T >::run().

```
00084         {
00085             timer = high\_resolution\_clock::now();
00086         }
```

6.11.3.3 stopTimer()

```
template<class T>
double mdata::SearchAlgorithm< T >::stopTimer ( ) [inline], [protected]
```

Returns the amount of time that has passed since [startTimer\(\)](#) was called in milliseconds.

Definition at line 91 of file [searchalg.h](#).

Referenced by [mdata::BlindSearch](#)< T >::run(), and [mdata::LocalSearch](#)< T >::run().

```
00092         {
00093             high\_resolution\_clock::time\_point t_end = high\_resolution\_clock::now();
00094             return static\_cast<double>(<a href="#">duration_cast<nanoseconds>(t_end - timer).count()) / 1000000.0;
00095         }
```

6.11.4 Member Data Documentation

6.11.4.1 timeDiff

```
template<class T>
double mdata::SearchAlgorithm< T >::timeDiff [protected]
```

Definition at line 77 of file [searchalg.h](#).

6.11.4.2 timer

```
template<class T>
high_resolution_clock::time_point mdata::SearchAlgorithm< T >::timer [protected]
```

Definition at line 78 of file [searchalg.h](#).

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

- include/[searchalg.h](#)

6.12 mdata::TestParameters< T > Struct Template Reference

Packs together various test experiment parameters.

```
#include <testparam.h>
```

Public Member Functions

- [TestParameters](#) ()

Public Attributes

- unsigned int [funcId](#)
- T [alpha](#)
- unsigned int [resultsCol](#)
- unsigned int [execTimesCol](#)
- size_t [resultsRow](#)
- size_t [execTimesRow](#)
- [DataTable](#)< T > * [resultsTable](#)
- [DataTable](#)< T > * [execTimesTable](#)
- [enums::Algorithm](#) [alg](#)

6.12.1 Detailed Description

```
template<class T>
struct mdata::TestParameters< T >
```

Packs together various test experiment parameters.

Template Parameters

<i>T</i>	The data type that the test algorithm and functions are using
----------	---

Definition at line 29 of file [testparam.h](#).

6.12.2 Constructor & Destructor Documentation

6.12.2.1 TestParameters()

```
template<class T>
mdata::TestParameters< T >::TestParameters ( ) [inline]
```

Selected search algorithm for this test

Definition at line 41 of file [testparam.h](#).

References [enums::BlindSearch](#).

```
00042     {
00043         funcId = 1;
00044         alpha = 0;
00045         alg = enums::Algorithm::BlindSearch;
00046         resultsTable = nullptr;
00047         execTimesTable = nullptr;
00048         resultsCol = 0;
00049         execTimesCol = 0;
00050         resultsRow = 0;
00051         execTimesRow = 0;
00052     }
```

6.12.3 Member Data Documentation

6.12.3.1 alg

```
template<class T>
enums::Algorithm mdata::TestParameters< T >::alg
```

Pointer to the [DataTable](#) used to store the execution times

Definition at line 39 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.2 alpha

```
template<class T>
T mdata::TestParameters< T >::alpha
```

Id for the tested math function

Definition at line 32 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.3 execTimesCol

```
template<class T>
unsigned int mdata::TestParameters< T >::execTimesCol
```

[DataTable](#) column index to store the fitness result

Definition at line 34 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.4 execTimesRow

```
template<class T>
size_t mdata::TestParameters< T >::execTimesRow
```

[DataTable](#) row index to store the fitness result

Definition at line 36 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.5 execTimesTable

```
template<class T>
DataTable<T>* mdata::TestParameters< T >::execTimesTable
```

Pointer to the [DataTable](#) used to store the fitness results

Definition at line 38 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.6 funcId

```
template<class T>
unsigned int mdata::TestParameters< T >::funcId
```

Definition at line 31 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.7 resultsCol

```
template<class T>
unsigned int mdata::TestParameters< T >::resultsCol
```

Alpha value if needed by the search alg

Definition at line 33 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.8 resultsRow

```
template<class T>
size_t mdata::TestParameters< T >::resultsRow
```

[DataTable](#) column index to store the execution time

Definition at line 35 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

6.12.3.9 resultsTable

```
template<class T>
DataTable<T>* mdata::TestParameters< T >::resultsTable
```

[DataTable](#) row index to store the execution time

Definition at line 37 of file [testparam.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [mfunc::Experiment< T >::testFuncThreaded\(\)](#).

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

- include/[testparam.h](#)

6.13 mdata::TestResult< T > Struct Template Reference

```
#include <testresult.h>
```

Public Member Functions

- [TestResult](#) (int `_err`, T `_fitness`, double `_execTime`)

Public Attributes

- const int `err`
- const T `fitness`
- const double `execTime`

6.13.1 Detailed Description

```
template<class T>  
struct mdata::TestResult< T >
```

Definition at line 20 of file [testresult.h](#).

6.13.2 Constructor & Destructor Documentation

6.13.2.1 TestResult()

```
template<class T>  
mdata::TestResult< T >::TestResult (   
    int _err,  
    T _fitness,  
    double _execTime ) [inline]
```

Definition at line 26 of file [testresult.h](#).

```
00026                                     : err (_err),  
    fitness (_fitness), execTime (_execTime)  
00027     {  
00028     }
```

6.13.3 Member Data Documentation

6.13.3.1 err

```
template<class T>
const int mdata::TestResult< T >::err
```

Definition at line 22 of file [testresult.h](#).

6.13.3.2 execTime

```
template<class T>
const double mdata::TestResult< T >::execTime
```

Definition at line 24 of file [testresult.h](#).

6.13.3.3 fitness

```
template<class T>
const T mdata::TestResult< T >::fitness
```

Definition at line 23 of file [testresult.h](#).

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

- include/[testresult.h](#)

6.14 ThreadPool Class Reference

```
#include <threadpool.h>
```

Public Member Functions

- [ThreadPool](#) (size_t)
- template<class F, class... Args>
auto [enqueue](#) (F &&f, Args &&... args) -> std::future< typename std::result_of< F(Args...)>::type >
- [~ThreadPool](#) ()
- void [stopAndJoinAll](#) ()

6.14.1 Detailed Description

Copyright (c) 2012 Jakob Progsch, Václav Zeman <https://github.com/progschj/ThreadPool>

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

This source file has been modified slightly by Andrew Dunn

Definition at line 42 of file [threadpool.h](#).

6.14.2 Constructor & Destructor Documentation

6.14.2.1 ThreadPool()

```
ThreadPool::ThreadPool (
    size_t threads ) [inline]
```

Definition at line 64 of file [threadpool.h](#).

```
00065     :   stop(false)
00066 {
00067     for(size_t i = 0; i<threads; ++i)
00068         workers.emplace_back(
00069             [this]
00070             {
00071                 for(;;)
00072                 {
00073                     std::function<void()> task;
00074
00075                     {
00076                         std::unique_lock<std::mutex> lock(this->queue_mutex);
00077                         this->condition.wait(lock,
00078                             [this]{ return this->stop || !this->tasks.empty(); });
00079                         if(this->stop && this->tasks.empty())
00080                             return;
00081                         task = std::move(this->tasks.front());
00082                         this->tasks.pop();
00083                     }
00084
00085                     task();
00086                 }
00087             }
00088     );
00089 }
```

6.14.2.2 ~ThreadPool()

ThreadPool::~~ThreadPool () [inline]

Definition at line 117 of file [threadpool.h](#).

References [stopAndJoinAll\(\)](#).

```
00118 {
00119     stopAndJoinAll();
00120 }
```

6.14.3 Member Function Documentation

6.14.3.1 enqueue()

```
template<class F , class... Args>
auto ThreadPool::enqueue (
    F && f,
    Args &&... args ) -> std::future<typename std::result_of<F(Args...)>::type>
```

Definition at line 93 of file [threadpool.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#).

```
00095 {
00096     using return_type = typename std::result_of<F(Args...)>::type;
00097
00098     auto task = std::make_shared< std::packaged_task<return_type()> >(
00099         std::bind(std::forward<F>(f), std::forward<Args>(args)...)
00100     );
00101
00102     std::future<return_type> res = task->get_future();
00103     {
00104         std::unique_lock<std::mutex> lock(queue_mutex);
00105
00106         // don't allow enqueueing after stopping the pool
00107         if(stop)
00108             throw std::runtime_error("enqueue on stopped ThreadPool");
00109
00110         tasks.emplace([task]() { (*task)(); });
00111     }
00112     condition.notify_one();
00113     return res;
00114 }
```

6.14.3.2 stopAndJoinAll()

void ThreadPool::stopAndJoinAll () [inline]

Definition at line 122 of file [threadpool.h](#).

Referenced by [mfunc::Experiment< T >::testAllFunc\(\)](#), and [~ThreadPool\(\)](#).

```
00123 {
00124     {
00125         std::unique_lock<std::mutex> lock(queue_mutex);
00126         stop = true;
00127     }
00128
00129     condition.notify_all();
00130     for(std::thread &worker: workers)
00131         worker.join();
00132 }
```

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

- [include/threadpool.h](#)

Chapter 7

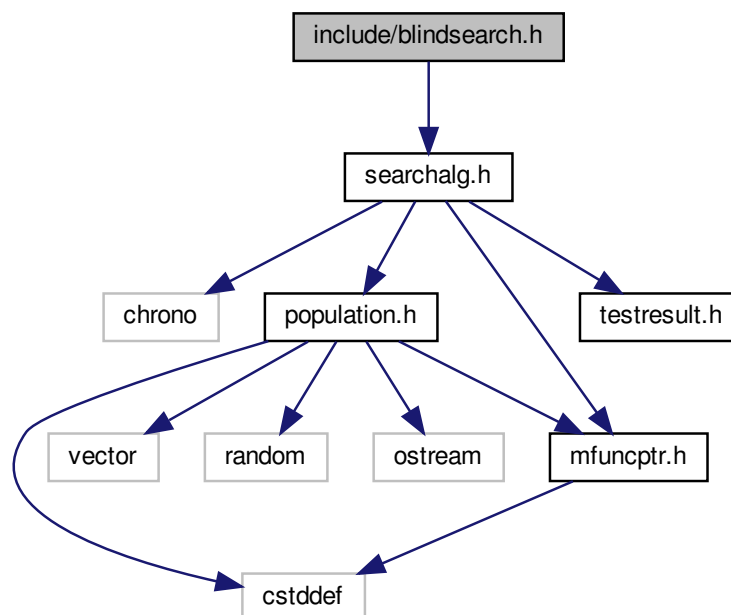
File Documentation

7.1 include/blindsearch.h File Reference

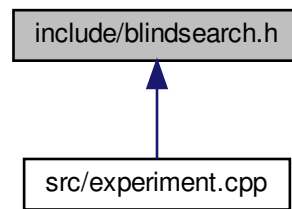
Implements the BlindSearch class, which inherits SearchAlgorithm. BlindSearch::run executes the blind search algorithm on a given population.

```
#include "searchalg.h"
```

Include dependency graph for blindsearch.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [mdata::BlindSearch< T >](#)

The [BlindSearch](#) class implements the Blind Search algorithm, which is ran using the overridden [SearchAlgorithm::run\(\)](#) function.

Namespaces

- [mdata](#)

7.1.1 Detailed Description

Implements the `BlindSearch` class, which inherits `SearchAlgorithm`. `BlindSearch::run` executes the blind search algorithm on a given population.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file [blindsearch.h](#).

7.2 blindsearch.h

```

00001
00013 #ifndef __BLINDSEARCH_H
00014 #define __BLINDSEARCH_H
00015
00016 #include "searchalg.h"
00017
00018 namespace mdata
00019 {
00026     template<class T>
00027     class BlindSearch : public SearchAlgorithm<T>
00028     {
00029         // Declaration needed due to template base class
00030         using SearchAlgorithm<T>::startTimer;
00031         using SearchAlgorithm<T>::stopTimer;
00032
00033     public:
00044         virtual TestResult<T> run(mfunc::mfuncPtr<T> funcPtr, const T
fMin, const T fMax, Population<T>* const pop, const T alpha)
00045         {
00046             // Get population size and dimensions
00047             size_t popSize = pop->getPopulationSize();
00048             size_t dimSize = pop->getDimensionsSize();
00049
00050             // Make sure funcPtr is valid;
00051             if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id,
return with error code 1
00052
00053             // Start recording execution time
00054             startTimer();
00055
00056             // Generate values for population vector matrix
00057             pop->generate(fMin, fMax);
00058
00059             // For each population vector, calculate the fitness using the funcPtr
00060             for (size_t sol = 0; sol < popSize; sol++)
00061             {
00062                 // Populate fitness values using given math function pointer
00063                 if (!pop->calcFitness(sol, funcPtr))
00064                     return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with
error code 2
00065             }
00066
00067             // Return best fitness value in population
00068             return TestResult<T>(0, *pop->getBestFitnessPtr(),
stopTimer());
00069         }
00070     };
00071 }
00072
00073 #endif
00074
00075 // =====
00076 // End of blindsearch.h
00077 // =====

```

7.3 include/datatable.h File Reference

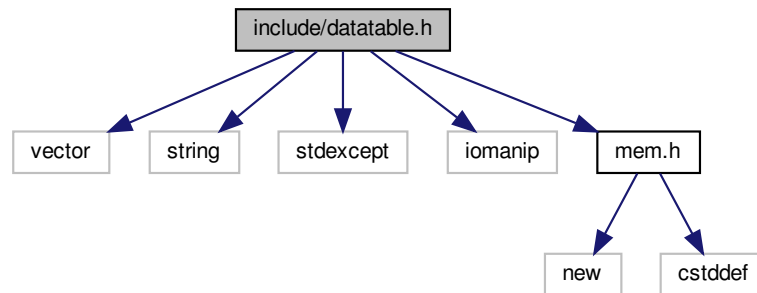
Header file for the DataTable class, which represents a spreadsheet/table of values that can easily be exported to a *.csv file.

```

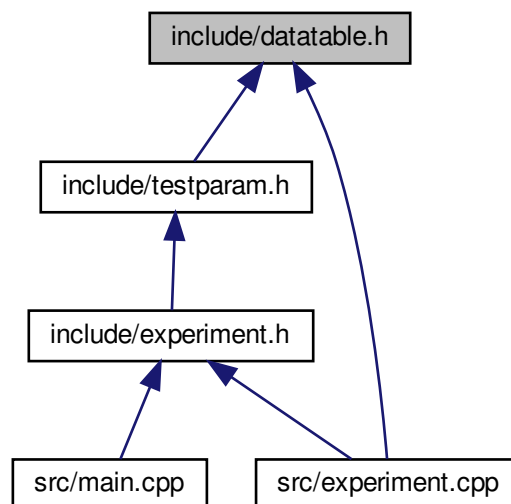
#include <vector>
#include <string>
#include <stdexcept>
#include <iomanip>
#include "mem.h"

```

Include dependency graph for datatable.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [mdata::DataTable< T >](#)

The [DataTable](#) class is a simple table of values with labeled columns.

Namespaces

- [mdata](#)

7.3.1 Detailed Description

Header file for the DataTable class, which represents a spreadsheet/table of values that can easily be exported to a *.csv file.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file [datatable.h](#).

7.4 datatable.h

```

00001
00013 #ifndef __DATATABLE_H
00014 #define __DATATABLE_H
00015
00016 #include <vector>
00017 #include <string>
00018 #include <stdexcept>
00019 #include <iomanip>
00020 #include "mem.h"
00021
00022 namespace mdata
00023 {
00049     template <class T>
00050     class DataTable
00051     {
00052     public:
00060         DataTable(size_t _rows, size_t _cols) : rows(_rows), cols(_cols), dataMatrix(nullptr)
00061         {
00062             if (rows == 0)
00063                 throw std::length_error("Table rows must be greater than 0.");
00064             else if (cols == 0)
00065                 throw std::length_error("Table columns must be greater than 0.");
00066
00067             dataMatrix = util::allocMatrix<T>(rows, cols);
00068             if (dataMatrix == nullptr)
00069                 throw std::bad_alloc();
00070
00071             colLabels.resize(_cols, std::string());
00072         }
00073
00077         ~DataTable()
00078         {
00079             util::releaseMatrix(dataMatrix, rows);
00080         }
00081
00088         std::string getColLabel(size_t colIndex)
00089         {
00090             if (colIndex >= colLabels.size())
00091                 throw std::out_of_range("Column index out of range");
00092
00093             return colLabels[colIndex];

```

```

00094     }
00095
00102 void setColLabel(size_t colIndex, std::string newLabel)
00103 {
00104     if (colIndex >= colLabels.size())
00105         throw std::out_of_range("Column index out of range");
00106
00107     colLabels[colIndex] = newLabel;
00108 }
00109
00117 T getEntry(size_t row, size_t col)
00118 {
00119     if (dataMatrix == nullptr)
00120         throw std::runtime_error("Data matrix not allocated");
00121     if (row >= rows)
00122         throw std::out_of_range("Table row out of range");
00123     else if (col >= cols)
00124         throw std::out_of_range("Table column out of range");
00125
00126     return dataMatrix[row][col];
00127 }
00128
00136 void setEntry(size_t row, size_t col, T val)
00137 {
00138     if (dataMatrix == nullptr)
00139         throw std::runtime_error("Data matrix not allocated");
00140     if (row >= rows)
00141         throw std::out_of_range("Table row out of range");
00142     else if (col >= cols)
00143         throw std::out_of_range("Table column out of range");
00144
00145     dataMatrix[row][col] = val;
00146 }
00147
00155 bool exportCSV(const char* filePath)
00156 {
00157     if (dataMatrix == nullptr) return false;
00158
00159     using namespace std;
00160     ofstream outFile;
00161     outFile.open(filePath, ofstream::out | ofstream::trunc);
00162     if (!outFile.good()) return false;
00163
00164     // Print column labels
00165     for (unsigned int c = 0; c < cols; c++)
00166     {
00167         outFile << colLabels[c];
00168         if (c < cols - 1) outFile << ",";
00169     }
00170
00171     outFile << endl;
00172
00173     // Print data rows
00174     for (unsigned int r = 0; r < rows; r++)
00175     {
00176         for (unsigned int c = 0; c < cols; c++)
00177         {
00178             outFile << std::setprecision(8) << dataMatrix[r][c];
00179             if (c < cols - 1) outFile << ",";
00180         }
00181         outFile << endl;
00182     }
00183
00184     outFile.close();
00185     return true;
00186 }
00187 private:
00188     size_t rows;
00189     size_t cols;
00190     std::vector<std::string> colLabels;
00191     T** dataMatrix;
00192 };
00193 } // mdata
00194
00195
00196 #endif
00197
00198 // =====
00199 // End of datatable.h
00200 // =====

```

7.5 include/experiment.h File Reference

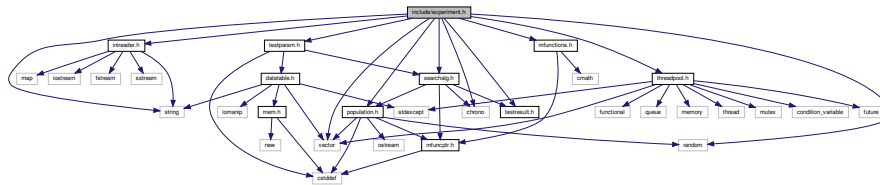
Header file for the Experiment class. Contains the basic logic and functions to run the cs471 project experiment.

```

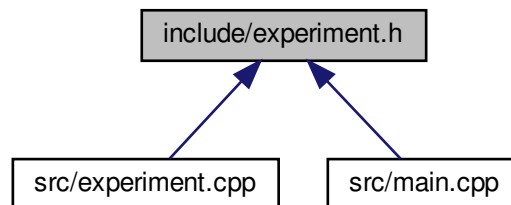
#include <string>
#include <random>
#include <chrono>
#include <vector>
#include "mfunctions.h"
#include "inireader.h"
#include "population.h"
#include "threadpool.h"
#include "searchalg.h"
#include "testresult.h"
#include "testparam.h"

```

Include dependency graph for experiment.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [mfunc::RandomBounds< T >](#)
Simple struct for storing the minimum and maximum input vector bounds for a function.
- class [mfunc::Experiment< T >](#)
Contains classes for running the CS471 project experiment.

Namespaces

- [mfunc](#)

7.5.1 Detailed Description

Header file for the Experiment class. Contains the basic logic and functions to run the cs471 project experiment.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file [experiment.h](#).

7.6 experiment.h

```

00001
00013 #ifndef __EXPERIMENT_H
00014 #define __EXPERIMENT_H
00015
00016 #include <string>
00017 #include <random>
00018 #include <chrono>
00019 #include <vector>
00020 #include "mfunctions.h"
00021 #include "inireader.h"
00022 #include "population.h"
00023 #include "threadpool.h"
00024 #include "searchalg.h"
00025 #include "testresult.h"
00026 #include "testparam.h"
00027
00028 namespace mfunc
00029 {
00034     template<class T>
00035     struct RandomBounds
00036     {
00037         T min = 0.0;
00038         T max = 0.0;
00039     };
00040
00051     template<class T>
00052     class Experiment
00053     {
00054     public:
00055         Experiment();
00056         ~Experiment();
00057         bool init(const char* paramFile);
00058         int testAllFunc();
00059         int testFuncThreaded(mdata::TestParameters<T> tParams);
00060     private:
00061         std::mutex popPoolMutex;
00062         util::IniReader iniParams;
00063         std::vector<mdata::Population<T>*> populationsPool;
00064         std::string resultsFile;
00065         std::string execTimesFile;
00066         RandomBounds<T>* vBounds;
00067         ThreadPool* tPool;

```

```

00068         size_t iterations;
00069         T alpha;
00070         enums::Algorithm testAlg;
00071
00072         mdata::Population<T>* popPoolRemove();
00073         void popPoolAdd(mdata::Population<T>* popPtr);
00074
00075         bool parseFuncBounds();
00076
00077         bool allocatePopulationPool(size_t count, size_t popSize, size_t dimensions);
00078         void releasePopulationPool();
00079
00080         bool allocateVBounds();
00081         void releaseVBounds();
00082
00083         bool allocateThreadPool(size_t numThreads);
00084         void releaseThreadPool();
00085     };
00086 } // mfunc
00087
00088 #endif
00089
00090 // =====
00091 // End of experiment.h
00092 // =====

```

7.7 include/inireader.h File Reference

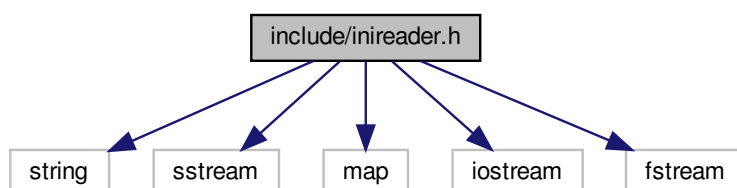
Header file for the IniReader class, which can open and parse simple *.ini files.

```

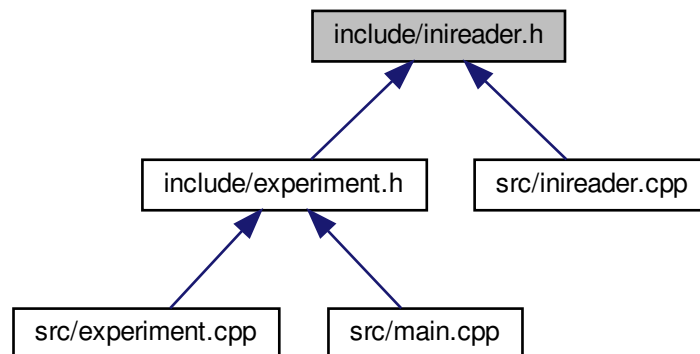
#include <string>
#include <sstream>
#include <map>
#include <iostream>
#include <fstream>

```

Include dependency graph for inireader.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [util::IniReader](#)
*The [IniReader](#) class is a simple *.ini file reader and parser.*

Namespaces

- [util](#)

7.7.1 Detailed Description

Header file for the IniReader class, which can open and parse simple *.ini files.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file [inireader.h](#).

7.8 inireader.h

```

00001
00013 #ifndef __INIREADER_H
00014 #define __INIREADER_H
00015
00016 #include <string>
00017 #include <sstream>
00018 #include <map>
00019 #include <iostream>
00020 #include <fstream>
00021
00022 namespace util
00023 {
00046     class IniReader
00047     {
00048     public:
00049         IniReader();
00050         ~IniReader();
00051         bool openFile(std::string filePath);
00052         bool sectionExists(std::string section);
00053         bool entryExists(std::string section, std::string entry);
00054         std::string getEntry(std::string section, std::string entry);
00055
00056         template <class T>
00057         T getEntryAs(std::string section, std::string entry)
00058         {
00059             std::stringstream ss(getEntry(section, entry));
00060             T retVal;
00061             ss >> retVal;
00062             return retVal;
00063         }
00064     private:
00065         std::string file;
00066         std::map<std::string, std::map<std::string, std::string>> iniMap;
00067         bool parseFile();
00068         void parseEntry(const std::string& sectionName, const std::string& entry);
00070     };
00071 }
00072
00073 #endif
00074
00075 // =====
00076 // End of inireader.h
00077 // =====

```

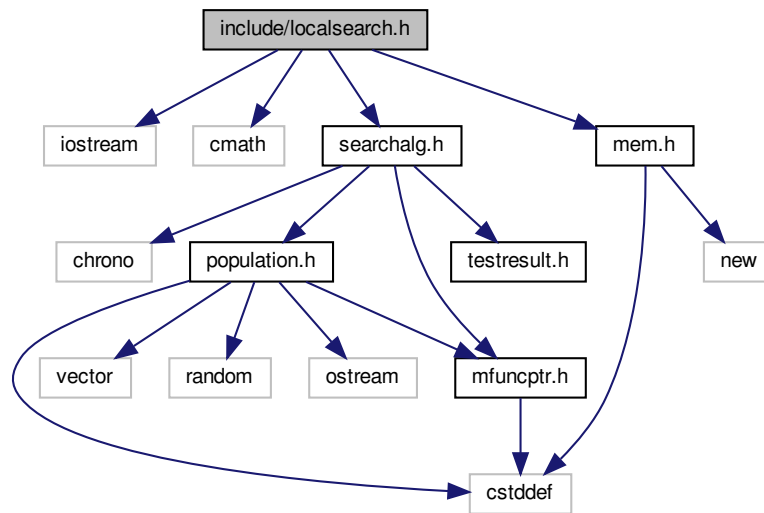
7.9 include/localsearch.h File Reference

```

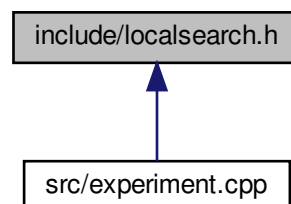
#include <iostream>
#include <cmath>
#include "searchalg.h"
#include "mem.h"

```

Include dependency graph for `localsearch.h`:



This graph shows which files directly or indirectly include this file:



Classes

- class `mdata::LocalSearch< T >`

The `LocalSearch` class implements the Local Search algorithm, which is ran using the overridden `SearchAlgorithm::run()` function.

Namespaces

- `mdata`

Macros

- `#define DEC_PRECISION 12`

7.9.1 Detailed Description

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file [localsearch.h](#).

7.9.2 Macro Definition Documentation

7.9.2.1 DEC_PRECISION

```
#define DEC_PRECISION 12
```

Definition at line 21 of file [localsearch.h](#).

7.10 localsearch.h

```
00001
00012 #ifndef __LOCALSEARCH_H
00013 #define __LOCALSEARCH_H
00014
00015 #include <iostream>
00016 #include <cmath>
00017 #include "searchalg.h"
00018 #include "mem.h"
00019
00020 // Precision when checking neighbor fitness in number of decimal places right side of zero.
00021 #define DEC_PRECISION 12
00022
00023 namespace mdata
00024 {
00031     template<class T>
00032     class LocalSearch : public SearchAlgorithm<T>
00033     {
00034     public:
00035         using SearchAlgorithm<T>::startTimer;
00036         using SearchAlgorithm<T>::stopTimer;
00037         const T MIN_IMPROVEMENT = pow(static_cast<T>(10), static_cast<T>(-1 *
00038         DEC_PRECISION));
00039
00040     public:
00049         virtual TestResult<T> run(mfunc::mfuncPtr<T> funcPtr, const T
00050         fMin, const T fMax, Population<T>* const pop, const T alpha)
00051         {
```

```

00051         // Get population size and dimensions
00052         const size_t popSize = pop->getPopulationSize();
00053         const size_t dimSize = pop->getDimensionsSize();
00054
00055         // Make sure funcPtr is valid;
00056         if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id,
return with error code 1
00057
00058         // Algorithm related variables
00059         bool stop = false;
00060         size_t pIndex = 0;
00061
00062         startTimer();
00063
00064         // Start recording execution time
00065         pop->generate(fMin, fMax);
00066
00067         for (size_t sol = 0; sol < popSize; sol++)
00068         {
00069             // Populate fitness values using given math function pointer
00070             if (!pop->calcFitness(sol, funcPtr))
00071                 return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with
error code 2
00072         }
00073
00074         // Get the index for the best fitness in the population
00075         pIndex = pop->getBestFitnessIndex();
00076
00077         // Get population vector and fitness of best solution
00078         T* x = pop->getPopulationPtr(pIndex);
00079         T xFit = pop->getFitness(pIndex);
00080
00081         // Create empty Y vector
00082         T* y = util::allocArray<T>(dimSize);
00083         T yFit = 0;
00084
00085         // Create empty Z vector
00086         T* z = util::allocArray<T>(dimSize);
00087         T zFit = 0;
00088
00089         if (x == nullptr || y == nullptr || z == nullptr)
00090         {
00091             std::cerr << "Error in Local Search: Memory allocation failed" << std::endl;
00092             return TestResult<T>(3, 0, 0.0);
00093         }
00094
00095         // Keep looping until search fails to improve
00096         while (!stop)
00097         {
00098             stop = true;
00099
00100             // Copy values from X vector into Y vector
00101             util::copyArray<T>(x, y, dimSize);
00102
00103             // Loop through each dimension in vector
00104             for (size_t a = 0; a < dimSize; a++)
00105             {
00106                 // Add alpha value to y[a]
00107                 y[a] = x[a] + alpha;
00108
00109                 // Make sure y[a] is within the function bounds
00110                 lockBounds(y[a], fMin, fMax);
00111
00112                 // Calculate fitness for y vector
00113                 yFit = funcPtr(y, dimSize);
00114
00115                 // Update Z[a] vector value based on the difference between Y and X
00116                 z[a] = x[a] - (alpha * (yFit - xFit));
00117
00118                 // Make sure z[a] is within the function bounds
00119                 lockBounds(z[a], fMin, fMax);
00120
00121                 y[a] = x[a]; // Reset y[a] to prepare for next loop
00122             }
00123
00124             zFit = funcPtr(z, dimSize);
00125
00126             // The following 'if' statement may cause extreme execution
00127             // times for some functions due to floating point precision:
00128             // if (zFit < xFit)
00129             //
00130             // The replacement 'if' statement below places a limit
00131             // on the minimum acknowledged improvement fitness,
00132             // hopefully preventing extreme run-times:
00133             if (xFit - zFit > MIN_IMPROVEMENT)
00134             {
00135                 // Z is an improvement on X, so keep searching

```

```

00136             stop = false;
00137
00138             // Swap Z and X for next loop
00139             T* tmp = x;
00140             x = z;
00141             xFit = zFit;
00142             z = tmp;
00143         }
00144     }
00145
00146     // Return best result
00147     return TestResult<T>(0, xFit, stopTimer());
00148 }
00149 private:
00150 void lockBounds(T& val, const T& min, const T& max)
00151 {
00152     if (val < min) val = min;
00153     else if (val > max) val = max;
00154 }
00155 };
00156 }
00157
00158 #endif
00159
00160 // =====
00161 // End of localssearch.h
00162 // =====

```

7.11 include/mem.h File Reference

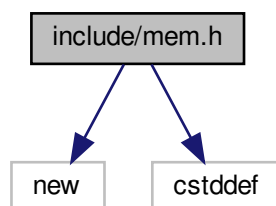
Header file for various memory utility functions.

```

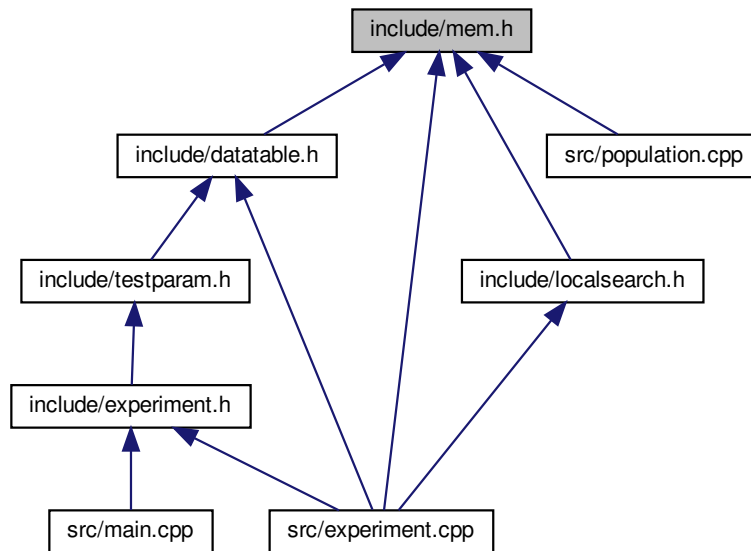
#include <new>
#include <cstddef>

```

Include dependency graph for mem.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- [util](#)

Functions

- `template<class T = double>`
`void util::initArray (T *a, size_t size, T val)`
Initializes an array with some set value.
- `template<class T = double>`
`void util::initMatrix (T **m, size_t rows, size_t cols, T val)`
Initializes a matrix with a set value for each entry.
- `template<class T = double>`
`void util::releaseArray (T *&a)`
Releases an allocated array's memory and sets the pointer to nullptr.
- `template<class T = double>`
`void util::releaseMatrix (T **&m, size_t rows)`
Releases an allocated matrix's memory and sets the pointer to nullptr.
- `template<class T = double>`
`T * util::allocArray (size_t size)`
Allocates a new array of the given data type.
- `template<class T = double>`
`T ** util::allocMatrix (size_t rows, size_t cols)`
Allocates a new matrix of the given data type.
- `template<class T = double>`
`void util::copyArray (T *src, T *dest, size_t size)`
Copies the elements from one equal-sized array to another.

7.11.1 Detailed Description

Header file for various memory utility functions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-02

Copyright

Copyright (c) 2019

Definition in file [mem.h](#).

7.12 mem.h

```

00001
00012 #ifndef __MEM_H
00013 #define __MEM_H
00014
00015 #include <new> // std::nothrow
00016 #include <cstdint> // size_t definition
00017
00018 namespace util
00019 {
00020     template <class T = double>
00021     inline void initArray(T* a, size_t size, T val)
00022     {
00023         if (a == nullptr) return;
00024         for (size_t i = 0; i < size; i++)
00025         {
00026             a[i] = val;
00027         }
00028     }
00029
00030     template <class T = double>
00031     inline void initMatrix(T** m, size_t rows, size_t cols, T val)
00032     {
00033         if (m == nullptr) return;
00034         for (size_t i = 0; i < rows; i++)
00035         {
00036             initArray(m[i], cols, val);
00037         }
00038     }
00039
00040     template <class T = double>
00041     void releaseArray(T*& a)
00042     {
00043         if (a == nullptr) return;
00044         delete[] a;
00045         a = nullptr;
00046     }
00047
00048     template <class T = double>
00049     void releaseMatrix(T**& m, size_t rows)

```

```

00083     {
00084         if (m == nullptr) return;
00085
00086         for (size_t i = 0; i < rows; i++)
00087         {
00088             if (m[i] != nullptr)
00089             {
00090                 // Release each row
00091                 releaseArray<T>(m[i]);
00092             }
00093         }
00094
00095         // Release columns
00096         delete[] m;
00097         m = nullptr;
00098     }
00099
00100 template <class T = double>
00101 inline T* allocArray(size_t size)
00102 {
00103     return new(std::nothrow) T[size];
00104 }
00105
00106 template <class T = double>
00107 inline T** allocMatrix(size_t rows, size_t cols)
00108 {
00109     T** m = (T**)allocArray<T*>(rows);
00110     if (m == nullptr) return nullptr;
00111
00112     for (size_t i = 0; i < rows; i++)
00113     {
00114         m[i] = allocArray<T>(cols);
00115         if (m[i] == nullptr)
00116         {
00117             releaseMatrix<T>(m, rows);
00118             return nullptr;
00119         }
00120     }
00121
00122     return m;
00123 }
00124
00125 template <class T = double>
00126 inline void copyArray(T* src, T* dest, size_t size)
00127 {
00128     for (size_t i = 0; i < size; i++)
00129         dest[i] = src[i];
00130 }
00131 }
00132
00133 #endif
00134
00135 // =====
00136 // End of mem.h
00137 // =====

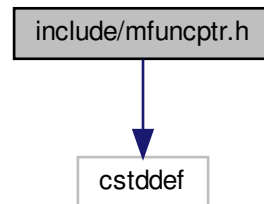
```

7.13 include/mfuncptr.h File Reference

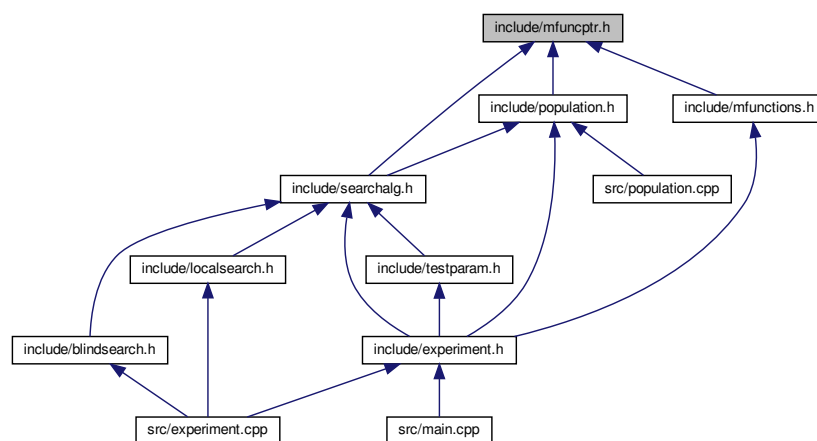
Contains the type definition for mfuncPtr, a templated function pointer to one of the math functions in [mfunctions.h](#).


```
#include <cstddef>
```

Include dependency graph for mfuncptr.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- [mfunc](#)

Typedefs

- `template<class T >`
`using mfunc::mfuncPtr = T(*)(T *, size_t)`

Function pointer that takes two arguments `T` and `size_t`, and returns a `T` value.*

7.13.1 Detailed Description

Contains the type definition for `mfuncPtr`, a templated function pointer to one of the math functions in [mfunctions.h](#).

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file [mfuncptr.h](#).

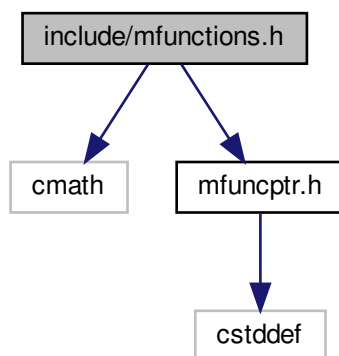
7.14 mfuncptr.h

```
00001
00014 #ifndef __MFUNCPTR_H
00015 #define __MFUNCPTR_H
00016
00017 #include <cstdint> // size_t definition
00018
00019 namespace mfunc
00020 {
00021     template <class T>
00028         using mfuncPtr = T (*)(T*, size_t);
00029 }
00030
00031 #endif
00032
00033 // =====
00034 // End of mfuncptr.h
00035 // =====
```

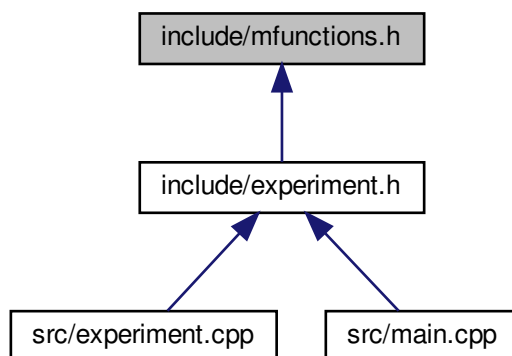
7.15 include/mfunctions.h File Reference

Contains various math function definitions.

```
#include <cmath>
#include "mfuncptr.h"
Include dependency graph for mfunctions.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- struct [mfunc::FunctionDesc](#)

[get\(\)](#) returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null

- struct [mfunc::Functions< T >](#)

Struct containing all static math functions. A function can be called directly by name, or indirectly using [Functions::get](#) or [Functions::exec](#).

Namespaces

- [mfunc](#)

Macros

- `#define _USE_MATH_DEFINES`
- `#define _schwefelDesc` "Schwefel's function"
- `#define _dejongDesc` "1st De Jong's function"
- `#define _rosenbrokDesc` "Rosenbrock"
- `#define _rastriginDesc` "Rastrigin"
- `#define _griewangkDesc` "Griewangk"
- `#define _sineEnvelopeSineWaveDesc` "Sine Envelope Sine Wave"
- `#define _stretchedVSineWaveDesc` "Stretched V Sine Wave"
- `#define _ackleysOneDesc` "Ackley's One"
- `#define _ackleysTwoDesc` "Ackley's Two"
- `#define _eggHolderDesc` "Egg Holder"
- `#define _ranaDesc` "Rana"
- `#define _pathologicalDesc` "Pathological"
- `#define _michalewiczDesc` "Michalewicz"
- `#define _mastersCosineWaveDesc` "Masters Cosine Wave"
- `#define _quarticDesc` "Quartic"
- `#define _levyDesc` "Levy"
- `#define _stepDesc` "Step"
- `#define _alpineDesc` "Alpine"

Variables

- `constexpr const unsigned int mfunc::NUM_FUNCTIONS = 18`

7.15.1 Detailed Description

Contains various math function definitions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-03-29

Copyright

Copyright (c) 2019

Definition in file [mfunctions.h](#).

7.15.2 Macro Definition Documentation

7.15.2.1 `_ackleysOneDesc`

```
#define _ackleysOneDesc "Ackley's One"
```

Definition at line 27 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.2 `_ackleysTwoDesc`

```
#define _ackleysTwoDesc "Ackley's Two"
```

Definition at line 28 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.3 `_alpineDesc`

```
#define _alpineDesc "Alpine"
```

Definition at line 37 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.4 `_dejongDesc`

```
#define _dejongDesc "1st De Jong's function"
```

Definition at line 21 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.5 `_eggHolderDesc`

```
#define _eggHolderDesc "Egg Holder"
```

Definition at line 29 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.6 `_griewangkDesc`

```
#define _griewangkDesc "Griewangk"
```

Definition at line 24 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.7 `_levyDesc`

```
#define _levyDesc "Levy"
```

Definition at line 35 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.8 `_mastersCosineWaveDesc`

```
#define _mastersCosineWaveDesc "Masters Cosine Wave"
```

Definition at line 33 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.9 `_michalewiczDesc`

```
#define _michalewiczDesc "Michalewicz"
```

Definition at line 32 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.10 `_pathologicalDesc`

```
#define _pathologicalDesc "Pathological"
```

Definition at line 31 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.11 `_quarticDesc`

```
#define _quarticDesc "Quartic"
```

Definition at line 34 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.12 `_ranaDesc`

```
#define _ranaDesc "Rana"
```

Definition at line 30 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.13 `_rastriginDesc`

```
#define _rastriginDesc "Rastrigin"
```

Definition at line 23 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.14 `_rosenbrokDesc`

```
#define _rosenbrokDesc "Rosenbrock"
```

Definition at line 22 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.15 `_schwefelDesc`

```
#define _schwefelDesc "Schwefel's function"
```

Definition at line 20 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.16 `_sineEnvelopeSineWaveDesc`

```
#define _sineEnvelopeSineWaveDesc "Sine Envelope Sine Wave"
```

Definition at line 25 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.17 `_stepDesc`

```
#define _stepDesc "Step"
```

Definition at line 36 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.18 `_stretchedVSineWaveDesc`

```
#define _stretchedVSineWaveDesc "Stretched V Sine Wave"
```

Definition at line 26 of file [mfunctions.h](#).

Referenced by [mfunc::FunctionDesc::get\(\)](#).

7.15.2.19 `_USE_MATH_DEFINES`

```
#define _USE_MATH_DEFINES
```

Definition at line 15 of file [mfunctions.h](#).

7.16 mfunctions.h

```

00001
00012 #ifndef __MFUNCTIONS_H
00013 #define __MFUNCTIONS_H
00014
00015 #define _USE_MATH_DEFINES
00016
00017 #include <cmath>
00018 #include "mfuncptr.h"
00019
00020 #define _schwefelDesc "Schwefel's function"
00021 #define _dejongDesc "1st De Jong's function"
00022 #define _rosenbrokDesc "Rosenbrock"
00023 #define _rastriginDesc "Rastrigin"
00024 #define _griewangkDesc "Griewangk"
00025 #define _sineEnvelopeSineWaveDesc "Sine Envelope Sine Wave"
00026 #define _stretchedVSineWaveDesc "Stretched V Sine Wave"
00027 #define _ackleysOneDesc "Ackley's One"
00028 #define _ackleysTwoDesc "Ackley's Two"
00029 #define _eggHolderDesc "Egg Holder"
00030 #define _ranaDesc "Rana"
00031 #define _pathologicalDesc "Pathological"
00032 #define _michalewiczDesc "Michalewicz"
00033 #define _mastersCosineWaveDesc "Masters Cosine Wave"
00034 #define _quarticDesc "Quartic"
00035 #define _levyDesc "Levy"
00036 #define _stepDesc "Step"
00037 #define _alpineDesc "Alpine"
00038
00042 namespace mfunc
00043 {
00047     constexpr const unsigned int NUM_FUNCTIONS = 18;
00048
00056     struct FunctionDesc
00057     {
00058         static const char* get(unsigned int f)
00059         {
00060             switch (f)
00061             {
00062                 case 1:
00063                     return _schwefelDesc;
00064                 case 2:
00065                     return _dejongDesc;
00066                 case 3:
00067                     return _rosenbrokDesc;
00068                 case 4:
00069                     return _rastriginDesc;
00070                 case 5:
00071                     return _griewangkDesc;
00072                 case 6:
00073                     return _sineEnvelopeSineWaveDesc;
00074                 case 7:
00075                     return _stretchedVSineWaveDesc;
00076                 case 8:
00077                     return _ackleysOneDesc;
00078                 case 9:
00079                     return _ackleysTwoDesc;
00080                 case 10:
00081                     return _eggHolderDesc;
00082                 case 11:
00083                     return _ranaDesc;
00084                 case 12:
00085                     return _pathologicalDesc;
00086                 case 13:
00087                     return _michalewiczDesc;
00088                 case 14:
00089                     return _mastersCosineWaveDesc;
00090                 case 15:
00091                     return _quarticDesc;
00092                 case 16:
00093                     return _levyDesc;
00094                 case 17:
00095                     return _stepDesc;
00096                 case 18:
00097                     return _alpineDesc;
00098                 default:
00099                     return NULL;
00100             }
00101         }
00102     };
00103
00111     template <class T>
00112     struct Functions
00113     {
00114         static T schwefel(T* v, size_t n);

```

```

00115     static T dejong(T* v, size_t n);
00116     static T rosenbrok(T* v, size_t n);
00117     static T rastrigin(T* v, size_t n);
00118     static T griewangk(T* v, size_t n);
00119     static T sineEnvelopeSineWave(T* v, size_t n);
00120     static T stretchedVSineWave(T* v, size_t n);
00121     static T ackleysOne(T* v, size_t n);
00122     static T ackleysTwo(T* v, size_t n);
00123     static T eggHolder(T* v, size_t n);
00124     static T rana(T* v, size_t n);
00125     static T pathological(T* v, size_t n);
00126     static T mastersCosineWave(T* v, size_t n);
00127     static T michalewicz(T* v, size_t n);
00128     static T quartic(T* v, size_t n);
00129     static T levy(T* v, size_t n);
00130     static T step(T* v, size_t n);
00131     static T alpine(T* v, size_t n);
00132     static mfuncPtr<T> get(unsigned int f);
00133     static bool exec(unsigned int f, T* v, size_t n, T& outResult);
00134     static T nthroot(T x, T n);
00135     static T w(T x);
00136 };
00137 }
00138
00145 template <class T>
00146 T mfunc::Functions<T>::nthroot(T x, T n)
00147 {
00148     return pow(x, static_cast<T>(1.0) / n);
00149 }
00150
00151 // =====
00152
00160 template <class T>
00161 T mfunc::Functions<T>::schwefel(T* v, size_t n)
00162 {
00163     T f = 0.0;
00164
00165     for (size_t i = 0; i < n; i++)
00166     {
00167         f += (static_cast<T>(-1.0) * v[i]) * sin(sqrt(std::abs(v[i])));
00168     }
00169
00170     return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
00171 }
00172
00173 // =====
00174
00182 template <class T>
00183 T mfunc::Functions<T>::dejong(T* v, size_t n)
00184 {
00185     T f = 0.0;
00186
00187     for (size_t i = 0; i < n; i++)
00188     {
00189         f += v[i] * v[i];
00190     }
00191
00192     return f;
00193 }
00194
00195 // =====
00196
00204 template <class T>
00205 T mfunc::Functions<T>::rosenbrok(T* v, size_t n)
00206 {
00207     T f = 0.0;
00208
00209     for (size_t i = 0; i < n - 1; i++)
00210     {
00211         T a = ((v[i] * v[i]) - v[i+1]);
00212         T b = (static_cast<T>(1.0) - v[i]);
00213         f += static_cast<T>(100.0) * a * a;
00214         f += b * b;
00215     }
00216
00217     return f;
00218 }
00219
00220 // =====
00221
00229 template <class T>
00230 T mfunc::Functions<T>::rastrigin(T* v, size_t n)
00231 {
00232     T f = 0.0;
00233
00234     for (size_t i = 0; i < n; i++)
00235     {

```

```

00236         f += (v[i] * v[i]) - (static_cast<T>(10.0) * cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i])
00237     );
00238     }
00239     return static_cast<T>(10.0) * static_cast<T>(n) * f;
00240 }
00241
00242 // =====
00243
00251 template <class T>
00252 T mfunc::Functions<T>::griewangk(T* v, size_t n)
00253 {
00254     T sum = 0.0;
00255     T product = 0.0;
00256
00257     for (size_t i = 0; i < n; i++)
00258     {
00259         sum += (v[i] * v[i]) / static_cast<T>(4000.0);
00260     }
00261
00262     for (size_t i = 0; i < n; i++)
00263     {
00264         product *= cos(v[i] / sqrt(static_cast<T>(i + 1.0)));
00265     }
00266
00267     return static_cast<T>(1.0) + sum - product;
00268 }
00269
00270 // =====
00271
00279 template <class T>
00280 T mfunc::Functions<T>::sineEnvelopeSineWave(T* v, size_t n)
00281 {
00282     T f = 0.0;
00283
00284     for (size_t i = 0; i < n - 1; i++)
00285     {
00286         T a = sin(v[i]*v[i] + v[i+1]*v[i+1] - static_cast<T>(0.5));
00287         a *= a;
00288         T b = (static_cast<T>(1.0) + static_cast<T>(0.001)*(v[i]*v[i] + v[i+1]*v[i+1]));
00289         b *= b;
00290         f += static_cast<T>(0.5) + (a / b);
00291     }
00292
00293     return static_cast<T>(-1.0) * f;
00294 }
00295
00296 // =====
00297
00305 template <class T>
00306 T mfunc::Functions<T>::stretchedVSineWave(T* v, size_t n)
00307 {
00308     T f = 0.0;
00309
00310     for (size_t i = 0; i < n - 1; i++)
00311     {
00312         T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
00313         T b = sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0)));
00314         b *= b;
00315         f += a * b + static_cast<T>(1.0);
00316     }
00317
00318     return f;
00319 }
00320
00321 // =====
00322
00330 template <class T>
00331 T mfunc::Functions<T>::ackleysOne(T* v, size_t n)
00332 {
00333     T f = 0.0;
00334
00335     for (size_t i = 0; i < n - 1; i++)
00336     {
00337         T a = (static_cast<T>(1.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2))) * sqrt(v[i]*v[i] + v[i+
00338 1]*v[i+1]);
00339         T b = static_cast<T>(3.0) * (cos(static_cast<T>(2.0) * v[i]) + sin(static_cast<T>(2.0) * v[i+1]));
00340         f += a + b;
00341     }
00342
00343     return f;
00344 }
00345
00346 // =====
00347
00354 template <class T>
00355 T mfunc::Functions<T>::ackleysTwo(T* v, size_t n)

```

```

00356 {
00357     T f = 0.0;
00358
00359     for (size_t i = 0; i < n - 1; i++)
00360     {
00361         T a = static_cast<T>(20.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2) * sqrt((v[i]*v[i] + v[i+1]
00362 ]*v[i+1]) / static_cast<T>(2.0)));
00363         T b = pow(static_cast<T>(M_E), static_cast<T>(0.5) *
00364 (cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + cos(static_cast<T>(2.0) *
00365 static_cast<T>(M_PI) * v[i+1])));
00366         f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
00367     }
00368     return f;
00369 }
00370 // =====
00371
00372 template <class T>
00373 T mfunc::Functions<T>::eggHolder(T* v, size_t n)
00374 {
00375     T f = 0.0;
00376
00377     for (size_t i = 0; i < n - 1; i++)
00378     {
00379         T a = static_cast<T>(-1.0) * v[i] * sin(sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0))));
00380         T b = (v[i+1] + static_cast<T>(47)) * sin(sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (v[i]/
00381 static_cast<T>(2.0))));
00382         f += a - b;
00383     }
00384     return f;
00385 }
00386 // =====
00387
00388 template <class T>
00389 T mfunc::Functions<T>::rana(T* v, size_t n)
00390 {
00391     T f = 0.0;
00392
00393     for (size_t i = 0; i < n - 1; i++)
00394     {
00395         T a = v[i] * sin(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * cos(sqrt(std::abs(v[i+1] +
00396 v[i] + static_cast<T>(1.0))));
00397         T b = (v[i+1] + static_cast<T>(1.0)) * cos(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) *
00398 sin(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00399         f += a + b;
00400     }
00401     return f;
00402 }
00403 // =====
00404
00405 template <class T>
00406 T mfunc::Functions<T>::pathological(T* v, size_t n)
00407 {
00408     T f = 0.0;
00409
00410     for (size_t i = 0; i < n - 1; i++)
00411     {
00412         T a = sin(sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
00413         a = (a*a) - static_cast<T>(0.5);
00414         T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
00415         b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
00416         f += static_cast<T>(0.5) + (a/b);
00417     }
00418     return f;
00419 }
00420 // =====
00421
00422 template <class T>
00423 T mfunc::Functions<T>::michalewicz(T* v, size_t n)
00424 {
00425     T f = 0.0;
00426
00427     for (size_t i = 0; i < n; i++)
00428     {
00429         f += sin(v[i]) * pow(sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)), static_cast<T>(20));
00430     }
00431     return -1.0 * f;
00432 }
00433 }
00434 // =====
00435
00436 template <class T>
00437 T mfunc::Functions<T>::michalewicz(T* v, size_t n)
00438 {
00439     T f = 0.0;
00440
00441     for (size_t i = 0; i < n; i++)
00442     {
00443         f += sin(v[i]) * pow(sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)), static_cast<T>(20));
00444     }
00445     return -1.0 * f;
00446 }
00447 }

```

```

00466 // =====
00467
00475 template <class T>
00476 T mfunc::Functions<T>::mastersCosineWave(T* v, size_t n)
00477 {
00478     T f = 0.0;
00479     for (size_t i = 0; i < n - 1; i++)
00480     {
00481         T a = pow(M_E, static_cast<T>(-1.0/8.0)*(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i+1]*v[i
00482     ]));
00483         T b = cos(static_cast<T>(4) * sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v[i+1]));
00484         f += a * b;
00485     }
00486
00487     return static_cast<T>(-1.0) * f;
00488 }
00489
00490 // =====
00491
00499 template <class T>
00500 T mfunc::Functions<T>::quartic(T* v, size_t n)
00501 {
00502     T f = 0.0;
00503
00504     for (size_t i = 0; i < n; i++)
00505     {
00506         f += (i+1) * v[i] * v[i] * v[i] * v[i];
00507     }
00508
00509     return f;
00510 }
00511
00512 // =====
00513
00517 template <class T>
00518 T mfunc::Functions<T>::w(T x)
00519 {
00520     return static_cast<T>(1.0) + (x - static_cast<T>(1.0)) / static_cast<T>(4.0);
00521 }
00522
00530 template <class T>
00531 T mfunc::Functions<T>::levy(T* v, size_t n)
00532 {
00533     T f = 0.0;
00534
00535     for (size_t i = 0; i < n - 1; i++)
00536     {
00537         T a = w(v[i]) - static_cast<T>(1.0);
00538         a *= a;
00539         T b = sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
00540         b *= b;
00541         T c = w(v[n - 1]) - static_cast<T>(1.0);
00542         c *= c;
00543         T d = sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
00544         d *= d;
00545         f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
00546     }
00547
00548     T e = sin(static_cast<T>(M_PI) * w(v[0]));
00549     return e*e + f;
00550 }
00551
00552 // =====
00553
00561 template <class T>
00562 T mfunc::Functions<T>::step(T* v, size_t n)
00563 {
00564     T f = 0.0;
00565
00566     for (size_t i = 0; i < n; i++)
00567     {
00568         T a = std::abs(v[i]) + static_cast<T>(0.5);
00569         f += a * a;
00570     }
00571
00572     return f;
00573 }
00574
00575 // =====
00576
00584 template <class T>
00585 T mfunc::Functions<T>::alpine(T* v, size_t n)
00586 {
00587     T f = 0.0;
00588
00589     for (size_t i = 0; i < n; i++)

```

```

00590     {
00591         f += std::abs(v[i] * sin(v[i]) + static_cast<T>(0.1)*v[i]);
00592     }
00593
00594     return f;
00595 }
00596
00597 // =====
00598
00600 template <class T>
00601 mfunc::mfuncPtr<T> mfunc::Functions<T>::get(unsigned int f)
00602 {
00603     switch (f)
00604     {
00605         case 1:
00606             return Functions<T>::schwefel;
00607         case 2:
00608             return Functions<T>::dejong;
00609         case 3:
00610             return Functions<T>::rosenbrok;
00611         case 4:
00612             return Functions<T>::rastrigin;
00613         case 5:
00614             return Functions<T>::griewangk;
00615         case 6:
00616             return Functions<T>::sineEnvelopeSineWave;
00617         case 7:
00618             return Functions<T>::stretchedVSineWave;
00619         case 8:
00620             return Functions<T>::ackleysOne;
00621         case 9:
00622             return Functions<T>::ackleysTwo;
00623         case 10:
00624             return Functions<T>::eggHolder;
00625         case 11:
00626             return Functions<T>::rana;
00627         case 12:
00628             return Functions<T>::pathological;
00629         case 13:
00630             return Functions<T>::michalewicz;
00631         case 14:
00632             return Functions<T>::mastersCosineWave;
00633         case 15:
00634             return Functions<T>::quartic;
00635         case 16:
00636             return Functions<T>::levy;
00637         case 17:
00638             return Functions<T>::step;
00639         case 18:
00640             return Functions<T>::alpine;
00641         default:
00642             return nullptr;
00643     }
00644 }
00645
00646 // =====
00647
00648 template <class T>
00649 bool mfunc::Functions<T>::exec(unsigned int f, T* v, size_t n, T& outResult)
00650 {
00651     auto fPtr = get(f);
00652     if (fPtr == nullptr) return false;
00653
00654     outResult = fPtr(v, n);
00655     return true;
00656 }
00657
00658 #endif
00659
00660 // =====
00661 // End of mfunctions.h
00662 // =====

```

7.17 include/population.h File Reference

Header file for the Population class. Stores a population and resulting fitness values.

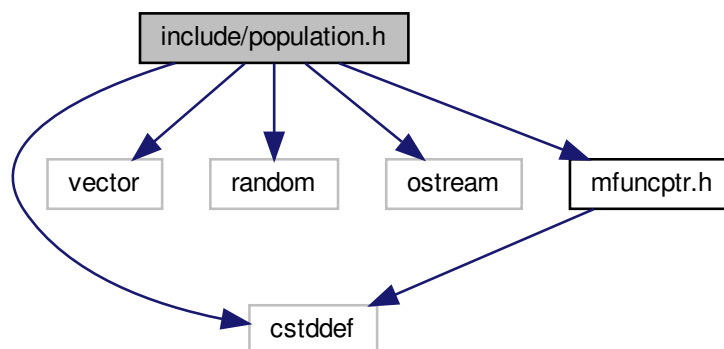
```

#include <cstdint>
#include <vector>

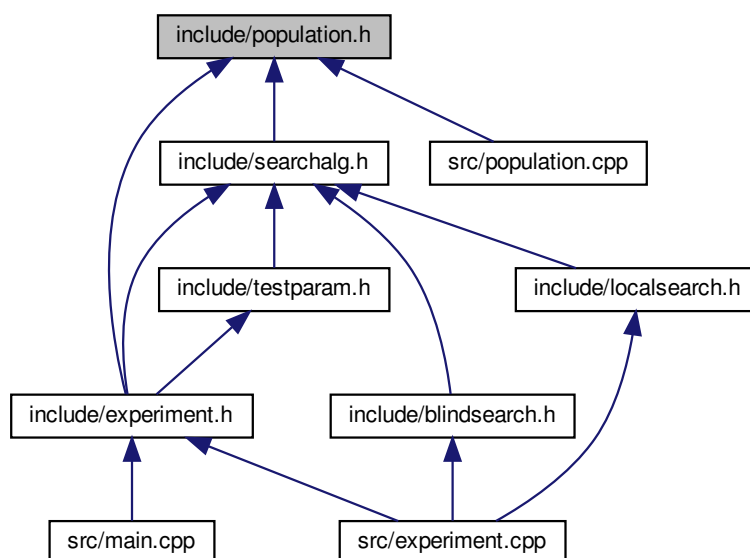
```

```
#include <random>
#include <ostream>
#include "mfuncptr.h"
```

Include dependency graph for population.h:



This graph shows which files directly or indirectly include this file:



Classes

- class `mdata::Population< T >`

Data class for storing a multi-dimensional population of data with the associated fitness.

Namespaces

- [mdata](#)

7.17.1 Detailed Description

Header file for the Population class. Stores a population and resulting fitness values.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-04

Copyright

Copyright (c) 2019

Definition in file [population.h](#).

7.18 population.h

```

00001
00012 #ifndef __POPULATION_H
00013 #define __POPULATION_H
00014
00015 #include <cstdint> // size_t definition
00016 #include <vector>
00017 #include <random>
00018 #include <ostream>
00019 #include "mfuncptr.h"
00020
00021 namespace mdata
00022 {
00023     template<class T>
00024     class Population
00025     {
00026     public:
00027         Population(size_t popSize, size_t dimensions);
00028         ~Population();
00029
00030         bool isReady();
00031         size_t getPopulationSize();
00032         size_t getDimensionsSize();
00033         T* getPopulationPtr(size_t popIndex);
00034
00035         bool generate(T minBound, T maxBound);
00036         bool setFitness(size_t popIndex, T value);
00037         bool calcFitness(size_t popIndex, mfunc::mfuncPtr<T> funcPtr);
00038
00039         T getFitness(size_t popIndex);
00040         T* getFitnessPtr(size_t popIndex);
00041         std::vector<T> getAllFitness();
00042         T* getBestFitnessPtr();
00043         size_t getBestFitnessIndex();
00044     };
00045 }
00046
00047
00048
00049
00050

```



```

00051     void outputPopulation(std::ostream& outStream, const char* delim, const char*
lineBreak);
00052     void outputFitness(std::ostream& outStream, const char* delim, const char* lineBreak);
00053     private:
00054         const size_t popSize;
00055         const size_t popDim;
00057         T** popMatrix;
00058         T* popFitness;
00060         std::random_device rdev;
00061         std::mt19937 rgen;
00063         bool allocPopMatrix();
00064         void releasePopMatrix();
00065
00066         bool allocPopFitness();
00067         void releasePopFitness();
00068     };
00069 }
00070
00071 #endif
00072
00073 // =====
00074 // End of population.h
00075 // =====

```

7.19 include/searchalg.h File Reference

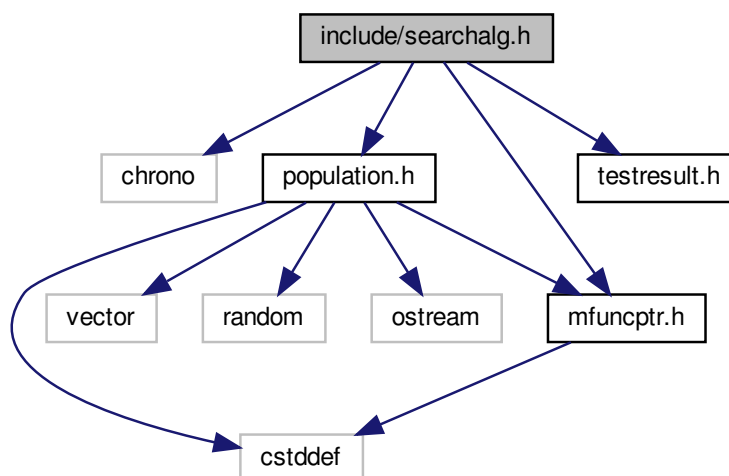
Defines the SearchAlgorithm class, Algorithm enum, and AlgorithmNames struct. The SearchAlgorithm class serves as a base class for implemented search algorithms.

```

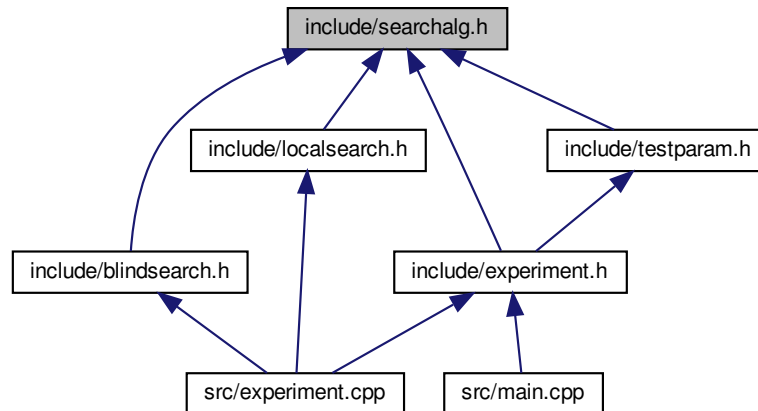
#include <chrono>
#include "population.h"
#include "testresult.h"
#include "mfuncptr.h"

```

Include dependency graph for searchalg.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [enums::AlgorithmNames](#)

Struct that contains constant string names for the different search algorithms.

- class [mdata::SearchAlgorithm< T >](#)

The [SearchAlgorithm](#) class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm.

Namespaces

- [enums](#)
- [mdata](#)

Enumerations

- enum [enums::Algorithm](#) { [enums::BlindSearch](#) = 0, [enums::LocalSearch](#) = 1, [enums::Count](#) = 2 }

Enum of different available search algorithms.

7.19.1 Detailed Description

Defines the SearchAlgorithm class, Algorithm enum, and AlgorithmNames struct. The SearchAlgorithm class serves as a base class for implemented search algorithms.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file [searchalg.h](#).

7.20 searchalg.h

```

00001
00013 #ifndef __SEARCHALG_H
00014 #define __SEARCHALG_H
00015
00016 #include <chrono>
00017 #include "population.h"
00018 #include "testresult.h"
00019 #include "mfuncptr.h"
00020
00021 using namespace std::chrono;
00022
00023 namespace enums
00024 {
00028     enum Algorithm
00029     {
00030         BlindSearch = 0,
00031         LocalSearch = 1,
00032         Count = 2
00033     };
00034
00039     struct AlgorithmNames
00040     {
00041         static constexpr const char* BLIND_SEARCH = "Blind Search";
00042         static constexpr const char* LOCAL_SEARCH = "Local Search";
00043
00044         static const char* get(Algorithm alg)
00045         {
00046             switch (alg)
00047             {
00048                 case Algorithm::BlindSearch:
00049                     return BLIND_SEARCH;
00050                 case Algorithm::LocalSearch:
00051                     return LOCAL_SEARCH;
00052                 default:
00053                     return "";
00054                 break;
00055             }
00056         }
00057     };
00058 }
00059
00060 namespace mdata
00061 {
00069     template<class T>
00070     class SearchAlgorithm
00071     {
00072     public:
00073         SearchAlgorithm() : timeDiff(0.0) {}
00074         virtual ~SearchAlgorithm() = 0;
00075         virtual TestResult<T> run(mfunc::mfuncPtr<T> funcPtr, const T fMin,
00076             const T fMax, Population<T>* pop, const T alpha) = 0;
00077     protected:
00077         double timeDiff;
00078         high_resolution_clock::time_point timer;
00079
00083         void startTimer()
00084         {
00085             timer = high_resolution_clock::now();

```

```

00086         }
00087
00091     double stopTimer()
00092     {
00093         high_resolution_clock::time_point t_end = high_resolution_clock::now();
00094         return static_cast<double>(duration_cast<nanoseconds>(t_end - timer).count()) / 1000000.0;
00095     }
00096 };
00097 }
00098
00099 // Trivial implementation of pure-virtual destructor
00100 // as required by the C++ language
00101 template<class T>
00102 mdata::SearchAlgorithm<T>::~SearchAlgorithm() { }
00103
00104 #endif
00105
00106 // =====
00107 // End of searchalg.h
00108 // =====

```

7.21 include/stringutils.h File Reference

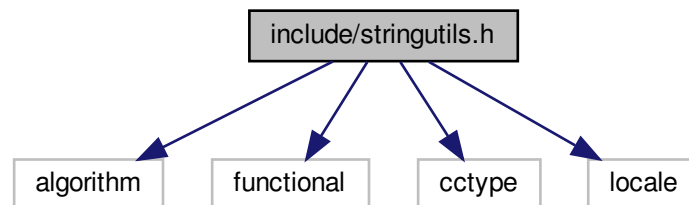
Contains various string manipulation helper functions.

```

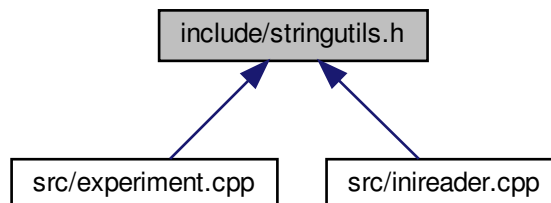
#include <algorithm>
#include <functional>
#include <cctype>
#include <locale>

```

Include dependency graph for stringutils.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- [util](#)

7.21.1 Detailed Description

Contains various string manipulation helper functions.

Author

Evan Teran (<https://github.com/eteran>)

Date

2019-04-01

Definition in file [stringutils.h](#).

7.22 stringutils.h

```

00001
00008 #ifndef __STRINGUTILS_H
00009 #define __STRINGUTILS_H
00010
00011 #include <algorithm>
00012 #include <functional>
00013 #include <cctype>
00014 #include <locale>
00015
00016 namespace util
00017 {
00018     // =====
00019     // The string functions below were written by Evan Teran
00020     // from Stack Overflow:
00021     // https://stackoverflow.com/questions/216823/whats-the-best-way-to-trim-stdstring
00022     // =====
00023
00024     // trim from start (in place)
00025     static inline void s_ltrim(std::string &s) {
00026         s.erase(s.begin(), std::find_if(s.begin(), s.end(),
00027             std::not1(std::ptr_fun<int, int>(std::isspace))));
00028     }
00029
00030     // trim from end (in place)
00031     static inline void s_rtrim(std::string &s) {
00032         s.erase(std::find_if(s.rbegin(), s.rend(),
00033             std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
00034     }
00035
00036     // trim from both ends (in place)
00037     static inline void s_trim(std::string &s) {
00038         s_ltrim(s);
00039         s_rtrim(s);
00040     }
00041
00042     // trim from start (copying)
00043     static inline std::string s_ltrim_copy(std::string s) {
00044         s_ltrim(s);
00045         return s;
00046     }
00047
00048     // trim from end (copying)
00049     static inline std::string s_rtrim_copy(std::string s) {
00050         s_rtrim(s);
00051         return s;
00052     }
00053
00054     // trim from both ends (copying)
00055     static inline std::string s_trim_copy(std::string s) {
00056         s_trim(s);
00057         return s;
00058     }
00059 }
00060 #endif
00061
00062 // =====
00063 // End of stringutils.h
00064 // =====

```

7.23 include/testparam.h File Reference

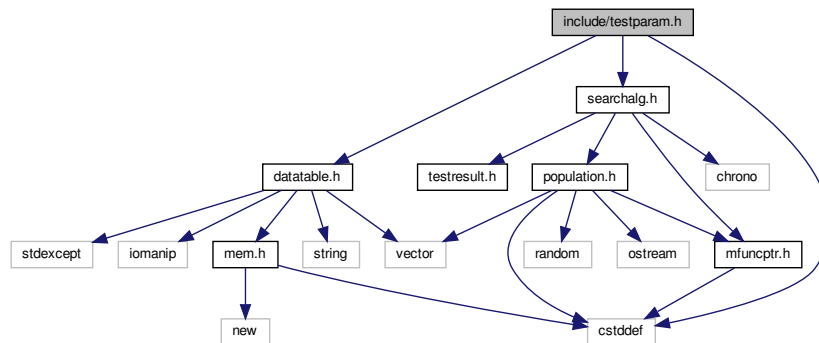
Contains the definition of the TestParameters struct, which is a data type used to transfer test parameters between functions.

```
#include <cstdint>
```

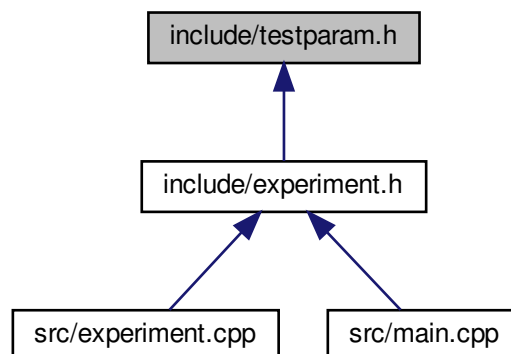
```
#include "datatable.h"
```

```
#include "searchalg.h"
```

Include dependency graph for testparam.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [mdata::TestParameters< T >](#)
Packs together various test experiment parameters.

Namespaces

- [mdata](#)

7.23.1 Detailed Description

Contains the definition of the TestParameters struct, which is a data type used to transfer test parameters between functions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-20

Copyright

Copyright (c) 2019

Definition in file [testparam.h](#).

7.24 testparam.h

```

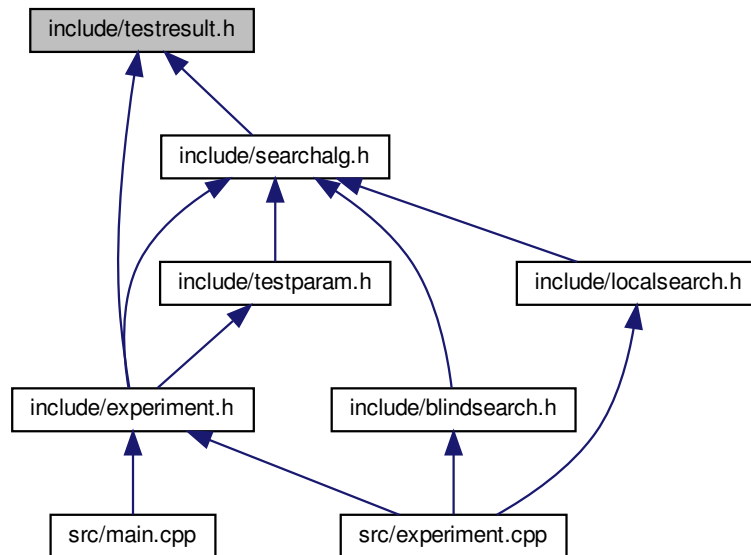
00001
00014 #ifndef __TESTPARAM_H
00015 #define __TESTPARAM_H
00016
00017 #include <cstdint> // size_t definition
00018 #include "datatable.h"
00019 #include "searchalg.h"
00020
00021 namespace mdata
00022 {
00028     template <class T>
00029     struct TestParameters
00030     {
00031         unsigned int funcId;
00032         T alpha;
00033         unsigned int resultsCol;
00034         unsigned int execTimesCol;
00035         size_t resultsRow;
00036         size_t execTimesRow;
00037         DataTable<T>* resultsTable;
00038         DataTable<T>* execTimesTable;
00039         enums::Algorithm alg;
00041         TestParameters()
00042         {
00043             funcId = 1;
00044             alpha = 0;
00045             alg = enums::Algorithm::BlindSearch;
00046             resultsTable = nullptr;
00047             execTimesTable = nullptr;
00048             resultsCol = 0;
00049             execTimesCol = 0;
00050             resultsRow = 0;
00051             execTimesRow = 0;
00052         }
00053     };
00054 }
00055
00056 #endif
00057
00058 // =====
00059 // End of testparam.h
00060 // =====

```

7.25 include/testresult.h File Reference

Simple structure that packs together various return values for the search algorithms. functions.

This graph shows which files directly or indirectly include this file:



Classes

- struct [mdata::TestResult< T >](#)

Namespaces

- [mdata](#)

7.25.1 Detailed Description

Simple structure that packs together various return values for the search algorithms. functions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file [testresult.h](#).

7.26 testresult.h

```

00001
00014 #ifndef __TESTRESULT_H
00015 #define __TESTRESULT_H
00016
00017 namespace mdata
00018 {
00019     template<class T>
00020     struct TestResult
00021     {
00022         const int err; // Error code. 0 = no error.
00023         const T fitness; // Fitness result
00024         const double execTime; // Algorithm execution time in milliseconds
00025
00026         TestResult(int _err, T _fitness, double _execTime) : err(_err), fitness(_fitness),
00027             execTime(_execTime)
00028         {
00029         };
00030     }; // mdata
00031
00032 #endif
00033
00034 // =====
00035 // End of testresult.h
00036 // =====

```

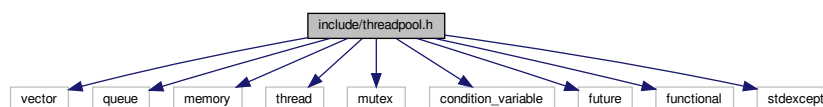
7.27 include/threadpool.h File Reference

```

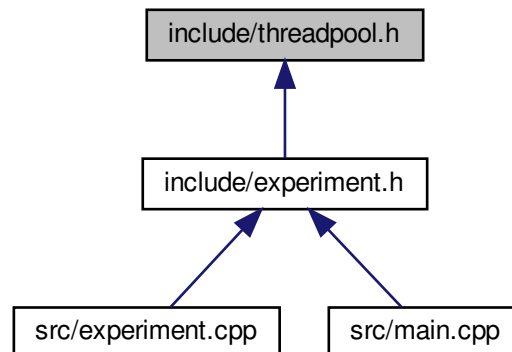
#include <vector>
#include <queue>
#include <memory>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <future>
#include <functional>
#include <stdexcept>

```

Include dependency graph for threadpool.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [ThreadPool](#)

7.28 threadpool.h

```

00001
00029 #ifndef __THREADPOOL_H
00030 #define __THREADPOOL_H
00031
00032 #include <vector>
00033 #include <queue>
00034 #include <memory>
00035 #include <thread>
00036 #include <mutex>
00037 #include <condition_variable>
00038 #include <future>
00039 #include <functional>
00040 #include <stdexcept>
00041
00042 class ThreadPool {
00043 public:
00044     ThreadPool(size_t);
00045     template<class F, class... Args>
00046     auto enqueue(F&& f, Args&&... args)
00047         -> std::future<typename std::result_of<F(Args...)>::type>;
00048     ~ThreadPool();
00049
00050     void stopAndJoinAll();
00051 private:
00052     // need to keep track of threads so we can join them
00053     std::vector<std::thread> workers;
00054     // the task queue
00055     std::queue<std::function<void()>> tasks;
00056
00057     // synchronization
00058     std::mutex queue_mutex;
00059     std::condition_variable condition;
00060     bool stop;
00061 };
00062
00063 // the constructor just launches some amount of workers
00064 inline ThreadPool::ThreadPool(size_t threads)
00065     : stop(false)
00066 {
00067     for(size_t i = 0; i<threads;++i)
  
```

```

00068         workers.emplace_back(
00069             [this]
00070             {
00071                 for(;;)
00072                 {
00073                     std::function<void()> task;
00074
00075                     {
00076                         std::unique_lock<std::mutex> lock(this->queue_mutex);
00077                         this->condition.wait(lock,
00078                             [this]{ return this->stop || !this->tasks.empty(); });
00079                         if(this->stop && this->tasks.empty())
00080                             return;
00081                         task = std::move(this->tasks.front());
00082                         this->tasks.pop();
00083                     }
00084
00085                     task();
00086                 }
00087             }
00088         );
00089     }
00090
00091     // add new work item to the pool
00092     template<class F, class... Args>
00093     auto ThreadPool::enqueue(F&& f, Args&&... args)
00094     -> std::future<typename std::result_of<F(Args...)>::type>
00095     {
00096         using return_type = typename std::result_of<F(Args...)>::type;
00097
00098         auto task = std::make_shared< std::packaged_task<return_type()> > (
00099             std::bind(std::forward<F>(f), std::forward<Args>(args)...)
00100         );
00101
00102         std::future<return_type> res = task->get_future();
00103         {
00104             std::unique_lock<std::mutex> lock(queue_mutex);
00105
00106             // don't allow enqueueing after stopping the pool
00107             if(stop)
00108                 throw std::runtime_error("enqueue on stopped ThreadPool");
00109
00110             tasks.emplace([task]() { (*task)(); });
00111         }
00112         condition.notify_one();
00113         return res;
00114     }
00115
00116     // the destructor joins all threads
00117     inline ThreadPool::~ThreadPool()
00118     {
00119         stopAndJoinAll();
00120     }
00121
00122     inline void ThreadPool::stopAndJoinAll()
00123     {
00124         {
00125             std::unique_lock<std::mutex> lock(queue_mutex);
00126             stop = true;
00127         }
00128
00129         condition.notify_all();
00130         for(std::thread &worker: workers)
00131             worker.join();
00132     }
00133
00134 #endif
00135
00136 // =====
00137 // End of threadpool.h
00138 // =====

```

7.29 src/experiment.cpp File Reference

```

#include <iostream>
#include <fstream>
#include <iomanip>
#include "experiment.h"
#include "datatable.h"

```


7.29.1.3 INI_TEST_ALPHA

```
#define INI_TEST_ALPHA "alpha"
```

Definition at line 30 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.4 INI_TEST_DIMENSIONS

```
#define INI_TEST_DIMENSIONS "dimensions"
```

Definition at line 27 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.5 INI_TEST_EXECTIONSFILE

```
#define INI_TEST_EXECTIONSFILE "exec_times_file"
```

Definition at line 33 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.6 INI_TEST_ITERATIONS

```
#define INI_TEST_ITERATIONS "iterations"
```

Definition at line 28 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.7 INI_TEST_NUMTHREADS

```
#define INI_TEST_NUMTHREADS "num_threads"
```

Definition at line 29 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.8 INI_TEST_POPULATION

```
#define INI_TEST_POPULATION "population"
```

Definition at line 26 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.9 INI_TEST_RESULTSFILE

```
#define INI_TEST_RESULTSFILE "results_file"
```

Definition at line 32 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.29.1.10 INI_TEST_SECTION

```
#define INI_TEST_SECTION "test"
```

Definition at line 24 of file [experiment.cpp](#).

Referenced by [mfunc::Experiment< T >::init\(\)](#).

7.30 experiment.cpp

```
00001
00013 #include <iostream>
00014 #include <fstream>
00015 #include <iomanip>
00016 #include "experiment.h"
00017 #include "datatable.h"
00018 #include "blindsearch.h"
00019 #include "localsearch.h"
00020 #include "stringutils.h"
00021 #include "mem.h"
00022
00023 // Ini file string sections and keys
00024 #define INI_TEST_SECTION "test"
00025 #define INI_FUNC_RANGE_SECTION "function_range"
00026 #define INI_TEST_POPULATION "population"
00027 #define INI_TEST_DIMENSIONS "dimensions"
00028 #define INI_TEST_ITERATIONS "iterations"
00029 #define INI_TEST_NUMTHREADS "num_threads"
00030 #define INI_TEST_ALPHA "alpha"
00031 #define INI_TEST_ALGORITHM "algorithm"
00032 #define INI_TEST_RESULTSFILE "results_file"
00033 #define INI_TEST_EXECTIONSFILE "exec_times_file"
00034
00035 using namespace std;
00036 using namespace std::chrono;
00037 using namespace mfunc;
00038
00042 template<class T>
00043 Experiment<T>::Experiment()
00044 : vBounds(nullptr), tPool(nullptr), resultsFile(""), execTimesFile(""), iterations(0)
00045 {
00046 }
00047
```

```

00052 template<class T>
00053 Experiment<T>::~Experiment()
00054 {
00055     releaseThreadPool();
00056     releasePopulationPool();
00057     releaseVBounds();
00058 }
00059
00060 template<class T>
00061 bool Experiment<T>::init(const char* paramFile)
00070 {
00071     try
00072     {
00073         // Open and parse parameters file
00074         if (!iniParams.openFile(paramFile))
00075         {
00076             cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
00077             return false;
00078         }
00079
00080         // Extract test parameters from ini file
00081         long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION,
00082 INI_TEST_POPULATION);
00083         long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION,
00084 INI_TEST_DIMENSIONS);
00085         long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION,
00086 INI_TEST_ITERATIONS);
00087         long numberThreads = iniParams.getEntryAs<long>(
00088 INI_TEST_SECTION, INI_TEST_NUMTHREADS);
00089         alpha = iniParams.getEntryAs<T>(INI_TEST_SECTION,
00090 INI_TEST_ALPHA);
00091         unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(
00092 INI_TEST_SECTION, INI_TEST_ALGORITHM);
00093         resultsFile = iniParams.getEntry(INI_TEST_SECTION,
00094 INI_TEST_RESULTSFILE);
00095         execTimesFile = iniParams.getEntry(INI_TEST_SECTION,
00096 INI_TEST_EXECTIMESFILE);
00097
00098         // Verify test parameters
00099         if (numberSol <= 0)
00100         {
00101             cerr << "Experiment init failed: Param file [test]->"
00102                 << INI_TEST_POPULATION << " entry missing or out of bounds: " <<
00103 paramFile << endl;
00104             return false;
00105         }
00106         else if (numberDim <= 0)
00107         {
00108             cerr << "Experiment init failed: Param file [test]->"
00109                 << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " <<
00110 paramFile << endl;
00111             return false;
00112         }
00113         else if (numberIter <= 0)
00114         {
00115             cerr << "Experiment init failed: Param file [test]->"
00116                 << INI_TEST_ITERATIONS << " entry missing or out of bounds: " <<
00117 paramFile << endl;
00118             return false;
00119         }
00120         else if (numberThreads <= 0)
00121         {
00122             cerr << "Experiment init failed: Param file [test]->"
00123                 << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " <<
00124 paramFile << endl;
00125             return false;
00126         }
00127         else if (alpha == 0)
00128         {
00129             cerr << "Experiment init failed: Param file [test]->"
00130                 << INI_TEST_ALPHA << " is missing or is equal to zero: " << paramFile << endl;
00131             return false;
00132         }
00133         else if (selectedAlg >= static_cast<unsigned int>(
00134 enums::Algorithm::Count))
00135         {
00136             cerr << "Experiment init failed: Param file [test]->"
00137                 << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile
00138                 << endl;
00139             return false;
00140         }
00141
00142         // Cast iterations and test algorithm to correct types
00143         iterations = (size_t)numberIter;
00144         testAlg = static_cast<enums::Algorithm>(selectedAlg);
00145     }
00146     catch (...)
00147     {
00148         cerr << "Experiment init failed: Unknown error" << endl;
00149         return false;
00150     }
00151 }

```

```

00132         // Print test parameters to console
00133         cout << "Population size: " << numberSol << endl;
00134         cout << "Dimensions: " << numberDim << endl;
00135         cout << "Iterations: " << iterations << endl;
00136         cout << "Alpha value: " << alpha << endl;
00137         cout << "Algorithm: " << enums::AlgorithmNames::get(testAlg) << endl;
00138
00139         // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
00140         if (!allocatePopulationPool((size_t)numberThreads, (size_t)numberSol, (size_t)numberDim))
00141         {
00142             cerr << "Experiment init failed: Unable to allocate populations." << endl;
00143             return false;
00144         }
00145
00146         // Allocate memory for function vector bounds
00147         if (!allocateVBounds())
00148         {
00149             cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
00150             return false;
00151         }
00152
00153         // Fill function bounds array with data parsed from iniParams
00154         if (!parseFuncBounds())
00155         {
00156             cerr << "Experiment init failed: Unable to parse vector bounds array." << endl;
00157             return false;
00158         }
00159
00160         // Allocate thread pool
00161         if (!allocateThreadPool((size_t)numberThreads))
00162         {
00163             cerr << "Experiment init failed: Unable to allocate thread pool." << endl;
00164             return false;
00165         }
00166
00167         cout << "Started " << numberThreads << " worker threads ..." << endl;
00168
00169         // Ready to run an experiment
00170         return true;
00171     }
00172     catch (const std::exception& ex)
00173     {
00174         cerr << "Exception occurred while initializing experiment: " << ex.what() << endl;
00175         return false;
00176     }
00177     catch (...)
00178     {
00179         cerr << "Unknown Exception occurred while initializing experiment." << endl;
00180         return false;
00181     }
00182 }
00183
00190 template<class T>
00191 int Experiment<T>::testAllFunc()
00192 {
00193     if (populationsPool.size() == 0) return 1;
00194
00195     // Construct results and execution times tables
00196     mdata::DataTable<T> resultsTable(iterations, (size_t)
NUM_FUNCTIONS);
00197     mdata::DataTable<T> execTimesTable(iterations, (size_t)NUM_FUNCTIONS);
00198
00199     // Prepare thread futures vector, used to ensure all async tasks complete
00200     // successfully.
00201     std::vector<std::future<int>> testFutures;
00202
00203     // Start recording total execution time
00204     high_resolution_clock::time_point t_start = high_resolution_clock::now();
00205
00206     // For each of the NUM_FUNCTIONS functions, prepare a TestParameters
00207     // struct and queue an asynchronous test that will be picked up and
00208     // executed by one of the threads in the thread pool.
00209     for (unsigned int i = 0; i < NUM_FUNCTIONS; i++)
00210     {
00211         // Update column labels for results and exec times tables
00212         resultsTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
00213         execTimesTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
00214
00215         // Queue up a new function test for each iteration
00216         for (size_t iter = 0; iter < iterations; iter++)
00217         {
00218             mdata::TestParameters<T> curParam;
00219             curParam.funcId = i + 1;
00220             curParam.alpha = alpha;
00221             curParam.alg = testAlg;
00222             curParam.resultsTable = &resultsTable;
00223             curParam.execTimesTable = &execTimesTable;

```



```

00224         curParam.resultsCol = i;
00225         curParam.execTimesCol = i;
00226         curParam.resultsRow = iter;
00227         curParam.execTimesRow = iter;
00228
00229         // Add function test to async queue
00230         testFutures.emplace_back(
00231             tPool->enqueue(&Experiment<T>::testFuncThreaded, this
, curParam)
00232         );
00233     }
00234 }
00235
00236 // Get the total number of async tasks queued
00237 const double totalFutures = static_cast<double>(testFutures.size());
00238 int tensPercentile = -1;
00239 std::chrono::microseconds waitTime(100);
00240
00241 // Loop until all async tasks are completed and the thread futures
00242 // array is empty
00243 while (testFutures.size() > 0)
00244 {
00245     // Sleep a little bit since the async thread tasks are higher priority
00246     std::this_thread::sleep_for(waitTime);
00247
00248     // Get iterator to first thread future
00249     auto it = testFutures.begin();
00250
00251     // Loop through all thread futures
00252     while (it != testFutures.end())
00253     {
00254         if (!it->valid())
00255         {
00256             // An error occurred with one of the threads
00257             cerr << "Error: Thread future invalid.";
00258             tPool->stopAndJoinAll();
00259             return 1;
00260         }
00261
00262         // Get the status of the current thread future (async task)
00263         std::future_status status = it->wait_for(waitTime);
00264         if (status == std::future_status::ready)
00265         {
00266             // Task has completed, get return value
00267             int errCode = it->get();
00268             if (errCode)
00269             {
00270                 // An error occurred while running the task.
00271                 // Bail out of function
00272                 tPool->stopAndJoinAll();
00273                 return errCode;
00274             }
00275
00276             // Remove processed task future from vector
00277             it = testFutures.erase(it);
00278
00279             // Calculate the percent completed of all tasks, rounded to the nearest 10%
00280             int curPercentile = static_cast<int>(((totalFutures - testFutures.size()) / totalFutures) *
10);
00281             if (curPercentile > tensPercentile)
00282             {
00283                 // Print latest percent value to the console
00284                 tensPercentile = curPercentile;
00285                 cout << "~" << (tensPercentile * 10) << "% " << flush;
00286             }
00287         }
00288         else
00289         {
00290             // Async task has not yet completed, advance to the next one
00291             it++;
00292         }
00293     }
00294 }
00295
00296 // Record total execution time and print it to the console
00297 high_resolution_clock::time_point t_end = high_resolution_clock::now();
00298 long double totalExecTime = static_cast<long double>(duration_cast<nanoseconds>(t_end - t_start).count()
) / 1000000000.0L;
00299
00300 cout << endl << "Test finished. Total time: " << std::setprecision(7) << totalExecTime << " seconds." <
< endl;
00301
00302 if (!resultsFile.empty())
00303 {
00304     // Export results table to a *.csv file
00305     cout << "Exporting results to: " << resultsFile << endl;
00306     resultsTable.exportCSV(resultsFile.c_str());

```

```

00307     }
00308
00309     if (!execTimesFile.empty())
00310     {
00311         // Export exec times table to a *.csv file
00312         cout << "Exporting execution times to: " << execTimesFile << endl;
00313         execTimesTable.exportCSV(execTimesFile.c_str());
00314     }
00315
00316     cout << flush;
00317
00318     return 0;
00319 }
00320
00321 template<class T>
00322 int Experiment<T>::testFuncThreaded(
00323     mdata::TestParameters<T> tParams)
00324 {
00325     mdata::SearchAlgorithm<T>* alg;
00326
00327     // Construct a search algorithm object for the selected alg
00328     switch (tParams.alg)
00329     {
00330     case enums::Algorithm::BlindSearch:
00331         alg = new mdata::BlindSearch<T>();
00332         break;
00333     case enums::Algorithm::LocalSearch:
00334         alg = new mdata::LocalSearch<T>();
00335         break;
00336     default:
00337         cerr << "Invalid algorithm selected." << endl;
00338         return 1;
00339     }
00340
00341     // Retrieve the function bounds
00342     const RandomBounds<T>& funcBounds = vBounds[tParams.funcId - 1];
00343
00344     // Retrieve the next available population object from the population pool
00345     mdata::Population<T>* pop = popPoolRemove();
00346
00347     // Run the search algorithm one and record the results
00348     auto tResult = alg->run(Functions<T>::get(tParams.funcId), funcBounds.
min, funcBounds.max, pop, tParams.alpha);
00349
00350     // Place the population object back into the pool to be reused by another thread
00351     popPoolAdd(pop);
00352
00353     if (tResult.err)
00354     {
00355         cerr << "Error while testing function " << tParams.funcId << endl;
00356         return tResult.err;
00357     }
00358
00359     // Update results table and execution times table with algorithm results
00360     tParams.resultsTable->setEntry(tParams.resultsRow, tParams.
resultsCol, tResult.fitness);
00361     tParams.execTimesTable->setEntry(tParams.execTimesRow, tParams.
execTimesCol, tResult.execTime);
00362
00363     delete alg;
00364     return 0;
00365 }
00366
00367 template<class T>
00368 mdata::Population<T>* Experiment<T>::popPoolRemove()
00369 {
00370     mdata::Population<T>* retPop = nullptr;
00371     std::chrono::microseconds waitTime(10);
00372
00373     while (true)
00374     {
00375         {
00376             std::lock_guard<std::mutex> lk(popPoolMutex);
00377             if (populationsPool.size() > 0)
00378             {
00379                 retPop = populationsPool.back();
00380                 populationsPool.pop_back();
00381             }
00382         }
00383
00384         if (retPop != nullptr)
00385             return retPop;
00386         else
00387             std::this_thread::sleep_for(waitTime);
00388     }
00389 }
00390
00391 }
00392
00393
00394
00395
00396
00397
00398
00399
00400
00401
00402
00403

```

```

00412 template<class T>
00413 void Experiment<T>::popPoolAdd(mdata::Population<T>* popPtr)
00414 {
00415     if (popPtr == nullptr) return;
00416
00417     std::lock_guard<std::mutex> lk(popPoolMutex);
00418
00419     populationsPool.push_back(popPtr);
00420 }
00421
00422 template<class T>
00423 bool Experiment<T>::parseFuncBounds()
00424 {
00425     if (vBounds == nullptr) return false;
00426
00427     const string delim = ",";
00428     const string section = "function_range";
00429     string s_min;
00430     string s_max;
00431
00432     // Extract the bounds for each function
00433     for (unsigned int i = 1; i <= NUM_FUNCTIONS; i++)
00434     {
00435         // Get bounds entry from ini file for current function
00436         string entry = iniParams.getEntry(section, to_string(i));
00437         if (entry.empty())
00438         {
00439             cerr << "Error parsing bounds for function: " << i << endl;
00440             return false;
00441         }
00442
00443         // Find index of ',' delimiter in entry string
00444         auto delimPos = entry.find(delim);
00445         if (delimPos == string::npos || delimPos >= entry.length() - 1)
00446         {
00447             cerr << "Error parsing bounds for function: " << i << endl;
00448             return false;
00449         }
00450
00451         // Split string and extract min/max strings
00452         s_min = entry.substr((size_t)0, delimPos);
00453         s_max = entry.substr(delimPos + 1, entry.length());
00454         util::s_trim(s_min);
00455         util::s_trim(s_max);
00456
00457         // Attempt to parse min and max strings into double values
00458         try
00459         {
00460             RandomBounds<T>& b = vBounds[i - 1];
00461             b.min = atof(s_min.c_str());
00462             b.max = atof(s_max.c_str());
00463         }
00464         catch(const std::exception& e)
00465         {
00466             cerr << "Error parsing bounds for function: " << i << endl;
00467             std::cerr << e.what() << '\n';
00468             return false;
00469         }
00470     }
00471
00472     return true;
00473 }
00474
00475 template<class T>
00476 bool Experiment<T>::allocatePopulationPool(size_t count, size_t
00477 popSize, size_t dimensions)
00478 {
00479     releasePopulationPool();
00480
00481     std::lock_guard<std::mutex> lk(popPoolMutex);
00482
00483     try
00484     {
00485         for (int i = 0; i < count; i++)
00486         {
00487             auto newPop = new(std::nothrow) mdata::Population<T>(popSize, dimensions);
00488             if (newPop == nullptr)
00489             {
00490                 std::cerr << "Error allocating populations." << '\n';
00491                 return false;
00492             }
00493
00494             populationsPool.push_back(newPop);
00495         }
00496
00497         return true;
00498     }
00499 }
00500
00501
00502
00503
00504
00505
00506
00507
00508
00509
00510

```

```

00511     catch(const std::exception& e)
00512     {
00513         std::cerr << e.what() << '\n';
00514         return false;
00515     }
00516 }
00517
00521 template<class T>
00522 void Experiment<T>::releasePopulationPool()
00523 {
00524     std::lock_guard<std::mutex> lk(popPoolMutex);
00525
00526     if (populationsPool.size() == 0) return;
00527
00528     for (int i = 0; i < populationsPool.size(); i++)
00529     {
00530         if (populationsPool[i] != nullptr)
00531         {
00532             delete populationsPool[i];
00533             populationsPool[i] = nullptr;
00534         }
00535     }
00536
00537     populationsPool.clear();
00538 }
00539
00547 template<class T>
00548 bool Experiment<T>::allocateVBounds()
00549 {
00550     vBounds = util::allocArray<RandomBounds<T>>(NUM_FUNCTIONS);
00551     return vBounds != nullptr;
00552 }
00553
00557 template<class T>
00558 void Experiment<T>::releaseVBounds()
00559 {
00560     if (vBounds == nullptr) return;
00561
00562     util::releaseArray<RandomBounds<T>>(vBounds);
00563 }
00564
00573 template<class T>
00574 bool Experiment<T>::allocateThreadPool(size_t numThreads)
00575 {
00576     releaseThreadPool();
00577
00578     tPool = new(std::nothrow) ThreadPool(numThreads);
00579     return tPool != nullptr;
00580 }
00581
00582 template<class T>
00583 void Experiment<T>::releaseThreadPool()
00584 {
00585     if (tPool == nullptr) return;
00586
00587     delete tPool;
00588     tPool = nullptr;
00589 }
00590
00591 // Explicit template specializations due to separate implementations in this CPP file
00592 template class mfunc::Experiment<float>;
00593 template class mfunc::Experiment<double>;
00594 template class mfunc::Experiment<long double>;
00595
00596 // =====
00597 // End of experiment.cpp
00598 // =====

```

7.31 src/inireader.cpp File Reference

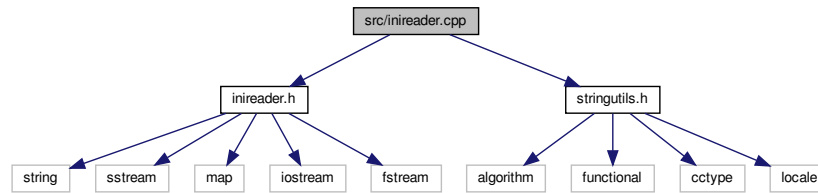
Implementation file for the IniReader class, which can open and parse simple *.ini files.

```

#include "inireader.h"
#include "stringutils.h"

```

Include dependency graph for inireader.cpp:



7.31.1 Detailed Description

Implementation file for the IniReader class, which can open and parse simple *.ini files.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file [inireader.cpp](#).

7.32 inireader.cpp

```

00001
00013 #include "inireader.h"
00014 #include "stringutils.h"
00015
00016 using namespace util;
00017
00021 IniReader::IniReader() : file(""), iniMap()
00022 {
00023 }
00024
00028 IniReader::~IniReader()
00029 {
00030     iniMap.clear();
00031 }
00032
00040 bool IniReader::openFile(std::string filePath)
00041 {
00042     file = filePath;
00043     if (!parseFile())
00044         return false;
  
```

```

00045
00046     return true;
00047 }
00048
00055 bool IniReader::sectionExists(std::string section)
00056 {
00057     return iniMap.find(section) != iniMap.end();
00058 }
00059
00067 bool IniReader::entryExists(std::string section, std::string entry)
00068 {
00069     auto it = iniMap.find(section);
00070     if (it == iniMap.end()) return false;
00071     return it->second.find(entry) != it->second.end();
00072 }
00073
00074
00084 std::string IniReader::getEntry(std::string section, std::string entry)
00085 {
00086     if (!entryExists(section, entry)) return std::string();
00087     return iniMap[section][entry];
00088 }
00089
00090
00097 bool IniReader::parseFile()
00098 {
00099     iniMap.clear();
00100
00101     using namespace std;
00102
00103     ifstream inputF(file, ifstream::in);
00104     if (!inputF.good()) return false;
00105
00106     string curSection;
00107     string line;
00108
00109     while (getline(inputF, line))
00110     {
00111         // Trim whitespace on both ends of the line
00112         s_trim(line);
00113
00114         // Ignore empty lines and comments
00115         if (line.empty() || line.front() == '#')
00116         {
00117             continue;
00118         }
00119         else if (line.front() == '[' && line.back() == ']')
00120         {
00121             // Line is a section definition
00122             // Erase brackets and trim to get section name
00123             line.erase(0, 1);
00124             line.erase(line.length() - 1, 1);
00125             s_trim(line);
00126             curSection = line;
00127         }
00128         else if (!curSection.empty())
00129         {
00130             // Line is an entry, parse the key and value
00131             parseEntry(curSection, line);
00132         }
00133     }
00134
00135     // Close input file
00136     inputF.close();
00137     return true;
00138 }
00139
00144 void IniReader::parseEntry(const std::string& sectionName, const std::string& entry)
00145 {
00146     using namespace std;
00147
00148     // Split string around equals sign character
00149     const string delim = "=";
00150     string entryName;
00151     string entryValue;
00152
00153     // Find index of '='
00154     auto delimPos = entry.find(delim);
00155
00156     if (delimPos == string::npos || delimPos >= entry.length() - 1)
00157         return; // '=' is missing, or is last char in string
00158
00159     // Extract entry name/key and value
00160     entryName = entry.substr((size_t)0, delimPos);
00161     entryValue = entry.substr(delimPos + 1, entry.length());
00162
00163     // Remove leading and trailing whitespace

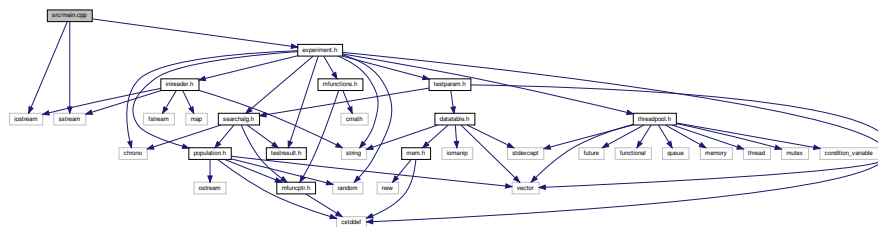
```

```
00164     s_trim(entryName);
00165     s_trim(entryValue);
00166
00167     // We cannot have entries with empty keys
00168     if (entryName.empty()) return;
00169
00170     // Add entry to cache
00171     iniMap[sectionName][entryName] = entryValue;
00172 }
00173
00174 // =====
00175 // End of inireader.cpp
00176 // =====
```

7.33 src/main.cpp File Reference

Program entry point. Creates and runs CS471 project 2 experiment.

```
#include <iostream>
#include <sstream>
#include "experiment.h"
Include dependency graph for main.cpp:
```



Functions

- `template<class T >`
`int runExp (const char *paramFile)`
Runs the experiment using the given data type and parameter file. Currently supports three different data types: float, double, and long double.
- `int main (int argc, char **argv)`

7.33.1 Detailed Description

Program entry point. Creates and runs CS471 project 2 experiment.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file [main.cpp](#).

7.33.2 Function Documentation

7.33.2.1 main()

```
int main (
    int argc,
    char ** argv )
```

Definition at line 46 of file [main.cpp](#).

```
00047 {
00048     // Make sure we have enough command line args
00049     if (argc <= 1)
00050     {
00051         cout << "Error: Missing command line parameter." << endl;
00052         cout << "Proper usage: " << argv[0] << " [param file]" << endl;
00053         return EXIT_FAILURE;
00054     }
00055     // Default data type is double
00056     int dataType = 1;
00057     // User specified a data type, retrieve the value
00058     if (argc > 2)
00059     {
00060         std::stringstream ss(argv[2]);
00061         ss >> dataType;
00062         if (!ss) dataType = 1;
00063     }
00064     // Verify specified data type switch
00065     if (dataType < 0 || dataType > 2)
00066     {
00067         cout << dataType << " is not a valid data type index. Value must be between 0 and 2." << endl;
00068         dataType = 1;
00069     }
00070     // Run experiment with correct data type and return success code
00071     switch (dataType)
00072     {
00073     case 0:
00074         return runExp<float>(argv[1]);
00075     case 1:
00076         return runExp<double>(argv[1]);
00077     case 2:
00078         return runExp<long double>(argv[1]);
00079     default:
00080         return EXIT_FAILURE;
00081     }
00082 }
00083 }
```

7.33.2.2 runExp()

```
template<class T >
int runExp (
    const char * paramFile )
```

Runs the experiment using the given data type and parameter file. Currently supports three different data types: float, double, and long double.

Template Parameters

<i>T</i>	
----------	--

Parameters

<i>paramFile</i>	
------------------	--

Returns

int

Definition at line 29 of file [main.cpp](#).References [mfunc::Experiment< T >::init\(\)](#), and [mfunc::Experiment< T >::testAllFunc\(\)](#).

```

00030 {
00031     // Create an instance of the project 1 experiment class
00032     mfunc::Experiment<T> ex;
00033
00034     // Print size of selected data type in bits
00035     cout << "Float size: " << (sizeof(T) * 8) << "-bits" << endl;
00036     cout << "Input parameters file: " << paramFile << endl;
00037     cout << "Initializing experiment ..." << endl;
00038
00039     // If experiment initialization fails, return failure
00040     if (!ex.init(paramFile))
00041         return EXIT_FAILURE;
00042     else
00043         return ex.testAllFunc();
00044 }

```

7.34 main.cpp

```

00001
00013 #include <iostream>
00014 #include <sstream>
00015 #include "experiment.h"
00016
00017 using namespace std;
00018
00028 template<class T>
00029 int runExp(const char* paramFile)
00030 {
00031     // Create an instance of the project 1 experiment class
00032     mfunc::Experiment<T> ex;
00033
00034     // Print size of selected data type in bits
00035     cout << "Float size: " << (sizeof(T) * 8) << "-bits" << endl;
00036     cout << "Input parameters file: " << paramFile << endl;
00037     cout << "Initializing experiment ..." << endl;
00038
00039     // If experiment initialization fails, return failure
00040     if (!ex.init(paramFile))
00041         return EXIT_FAILURE;
00042     else
00043         return ex.testAllFunc();
00044 }
00045
00046 int main(int argc, char** argv)
00047 {
00048     // Make sure we have enough command line args
00049     if (argc <= 1)
00050     {
00051         cout << "Error: Missing command line parameter." << endl;
00052         cout << "Proper usage: " << argv[0] << " [param file]" << endl;
00053         return EXIT_FAILURE;
00054     }
00055 }

```

```

00056 // Default data type is double
00057 int dataType = 1;
00058
00059 // User specified a data type, retrieve the value
00060 if (argc > 2)
00061 {
00062     std::stringstream ss(argv[2]);
00063     ss >> dataType;
00064     if (!ss) dataType = 1;
00065 }
00066
00067 // Verify specified data type switch
00068 if (dataType < 0 || dataType > 2)
00069 {
00070     cout << dataType << " is not a valid data type index. Value must be between 0 and 2." << endl;
00071     dataType = 1;
00072 }
00073
00074 // Run experiment with correct data type and return success code
00075 switch (dataType)
00076 {
00077     case 0:
00078         return runExp<float>(argv[1]);
00079     case 1:
00080         return runExp<double>(argv[1]);
00081     case 2:
00082         return runExp<long double>(argv[1]);
00083     default:
00084         return EXIT_FAILURE;
00085 }
00086 }
00087
00088 // =====
00089 // End of main.cpp
00090 // =====

```

7.35 src/population.cpp File Reference

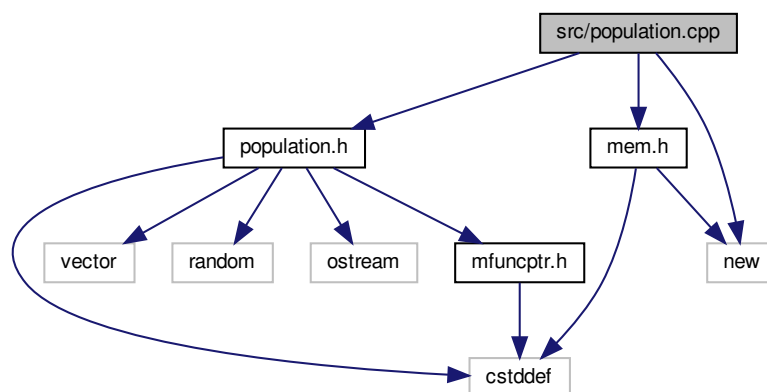
Implementation file for the Population class. Stores a population and fitness values.

```

#include "population.h"
#include "mem.h"
#include <new>

```

Include dependency graph for population.cpp:



7.35.1 Detailed Description

Implementation file for the Population class. Stores a population and fitness values.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-04

Copyright

Copyright (c) 2019

Definition in file [population.cpp](#).

7.36 population.cpp

```

00001
00012 #include "population.h"
00013 #include "mem.h"
00014 #include <new>
00015
00016 using namespace mdata;
00017 using namespace util;
00018
00026 template <class T>
00027 Population<T>::Population(size_t pSize, size_t dimensions) : popMatrix(nullptr),
    popSize(pSize), popDim(dimensions)
00028 {
00029     if (!allocPopMatrix() || !allocPopFitness())
00030         throw std::bad_alloc();
00031 }
00032
00038 template <class T>
00039 Population<T>::~Population()
00040 {
00041     releasePopMatrix();
00042     releasePopFitness();
00043 }
00044
00052 template <class T>
00053 bool Population<T>::isReady()
00054 {
00055     return popMatrix != nullptr && popFitness != nullptr;
00056 }
00057
00064 template <class T>
00065 size_t Population<T>::getPopulationSize()
00066 {
00067     return popSize;
00068 }
00069
00076 template <class T>
00077 size_t Population<T>::getDimensionsSize()
00078 {
00079     return popDim;
00080 }
00081
00089 template <class T>
00090 T* Population<T>::getPopulationPtr(size_t popIndex)
00091 {
00092     if (popFitness == nullptr || popIndex >= popSize) return nullptr;
00093     return popMatrix[popIndex];
00094 }
00095 }
00096
00107 template <class T>

```

```

00108 bool Population<T>::generate(T minBound, T maxBound)
00109 {
00110     if (popMatrix == nullptr) return false;
00111     // Generate a new seed for the mersenne twister engine
00112     rgen = std::mt19937(rdev());
00113     // Set up a normal (bell-shaped) distribution for the random number generator with the correct function
00114     bounds
00115     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00116     // Generate values for all vectors in popMatrix
00117     for (size_t s = 0; s < popSize; s++)
00118     {
00119         for (size_t d = 0; d < popDim; d++)
00120         {
00121             T rand = (T)dist(rgen);
00122             popMatrix[s][d] = rand;
00123         }
00124     }
00125     // Reset popFitness values to 0
00126     initArray<T>(popFitness, popSize, (T)0.0);
00127     return true;
00128 }
00129
00130 template<class T>
00131 bool Population<T>::setFitness(size_t popIndex, T value)
00132 {
00133     if (popFitness == nullptr || popIndex >= popSize) return false;
00134     popFitness[popIndex] = value;
00135     return true;
00136 }
00137
00138 template<class T>
00139 bool Population<T>::calcFitness(size_t popIndex,
00140     mfunc::mfuncPtr<T> funcPtr)
00141 {
00142     if (popFitness == nullptr || popIndex >= popSize) return false;
00143     popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);
00144     return true;
00145 }
00146
00147 template<class T>
00148 T Population<T>::getFitness(size_t popIndex)
00149 {
00150     if (popFitness == nullptr || popIndex >= popSize) return 0;
00151     return popFitness[popIndex];
00152 }
00153
00154 template<class T>
00155 T* Population<T>::getFitnessPtr(size_t popIndex)
00156 {
00157     if (popFitness == nullptr || popIndex >= popSize) return 0;
00158     return &popFitness[popIndex];
00159 }
00160
00161 template<class T>
00162 std::vector<T> Population<T>::getAllFitness()
00163 {
00164     return std::vector<T>(popFitness[0], popFitness[popSize]);
00165 }
00166
00167 template<class T>
00168 T* Population<T>::getBestFitnessPtr()
00169 {
00170     return &popFitness[getBestFitnessIndex()];
00171 }
00172
00173 template<class T>
00174 size_t Population<T>::getBestFitnessIndex()
00175 {
00176     size_t bestIndex = 0;
00177     for (size_t i = 1; i < popSize; i++)
00178     {
00179         if (popFitness[i] < popFitness[bestIndex])
00180             bestIndex = i;
00181     }
00182 }

```

```

00243     return bestIndex;
00244 }
00245
00254 template<class T>
00255 void Population<T>::outputPopulation(std::ostream& outStream, const char*
    delim, const char* lineBreak)
00256 {
00257     if (popMatrix == nullptr) return;
00258
00259     for (size_t j = 0; j < popSize; j++)
00260     {
00261         for (size_t k = 0; k < popDim; k++)
00262         {
00263             outStream << popMatrix[j][k];
00264             if (k < popDim - 1)
00265                 outStream << delim;
00266         }
00267         outStream << lineBreak;
00268     }
00269 }
00270 }
00271
00280 template<class T>
00281 void Population<T>::outputFitness(std::ostream& outStream, const char* delim,
    const char* lineBreak)
00282 {
00283     if (popFitness == nullptr) return;
00284
00285     for (size_t j = 0; j < popSize; j++)
00286     {
00287         outStream << popFitness[j];
00288         if (j < popSize - 1)
00289             outStream << delim;
00290     }
00291
00292     if (lineBreak != nullptr)
00293         outStream << lineBreak;
00294 }
00295
00302 template <class T>
00303 bool Population<T>::allocPopMatrix()
00304 {
00305     if (popSize == 0 || popDim == 0) return false;
00306
00307     popMatrix = allocMatrix<T>(popSize, popDim);
00308     initMatrix<T>(popMatrix, popSize, popDim, 0);
00309
00310     return popMatrix != nullptr;
00311 }
00312
00318 template <class T>
00319 void Population<T>::releasePopMatrix()
00320 {
00321     releaseMatrix<T>(popMatrix, popSize);
00322 }
00323
00330 template <class T>
00331 bool Population<T>::allocPopFitness()
00332 {
00333     if (popSize == 0 || popDim == 0) return false;
00334
00335     popFitness = allocArray<T>(popSize);
00336     initArray<T>(popFitness, popSize, 0);
00337
00338     return popFitness != nullptr;
00339 }
00340
00346 template <class T>
00347 void Population<T>::releasePopFitness()
00348 {
00349     releaseArray<T>(popFitness);
00350 }
00351
00352 // Explicit template specializations due to separate implementations in this CPP file
00353 template class mdata::Population<float>;
00354 template class mdata::Population<double>;
00355 template class mdata::Population<long double>;
00356
00357 // =====
00358 // End of population.cpp
00359 // =====

```


Index

- `_USE_MATH_DEFINES`
 - `mfunctions.h`, [108](#)
- `_ackleysOneDesc`
 - `mfunctions.h`, [105](#)
- `_ackleysTwoDesc`
 - `mfunctions.h`, [105](#)
- `_alpineDesc`
 - `mfunctions.h`, [105](#)
- `_dejongDesc`
 - `mfunctions.h`, [105](#)
- `_eggHolderDesc`
 - `mfunctions.h`, [105](#)
- `_griewangkDesc`
 - `mfunctions.h`, [105](#)
- `_levyDesc`
 - `mfunctions.h`, [106](#)
- `_mastersCosineWaveDesc`
 - `mfunctions.h`, [106](#)
- `_michalewiczDesc`
 - `mfunctions.h`, [106](#)
- `_pathologicalDesc`
 - `mfunctions.h`, [106](#)
- `_quarticDesc`
 - `mfunctions.h`, [106](#)
- `_ranaDesc`
 - `mfunctions.h`, [107](#)
- `_rastriginDesc`
 - `mfunctions.h`, [107](#)
- `_rosenbrokDesc`
 - `mfunctions.h`, [107](#)
- `_schwefelDesc`
 - `mfunctions.h`, [107](#)
- `_sineEnvelopeSineWaveDesc`
 - `mfunctions.h`, [107](#)
- `_stepDesc`
 - `mfunctions.h`, [108](#)
- `_stretchedVSineWaveDesc`
 - `mfunctions.h`, [108](#)
- `~DataTable`
 - `mdata::DataTable`, [22](#)
- `~Experiment`
 - `mfunc::Experiment`, [27](#)
- `~IniReader`
 - `util::IniReader`, [52](#)
- `~Population`
 - `mdata::Population`, [60](#)
- `~SearchAlgorithm`
 - `mdata::SearchAlgorithm`, [72](#)
- `~ThreadPool`
 - `ThreadPool`, [80](#)
- `ackleysOne`
 - `mfunc::Functions`, [36](#)
- `ackleysTwo`
 - `mfunc::Functions`, [37](#)
- `alg`
 - `mdata::TestParameters`, [75](#)
- `Algorithm`
 - `enums`, [9](#)
- `allocArray`
 - `util`, [12](#)
- `allocMatrix`
 - `util`, [13](#)
- `alpha`
 - `mdata::TestParameters`, [75](#)
- `alpine`
 - `mfunc::Functions`, [37](#)
- `BLIND_SEARCH`
 - `enums::AlgorithmNames`, [18](#)
- `calcFitness`
 - `mdata::Population`, [60](#)
- `copyArray`
 - `util`, [13](#)
- `Count`
 - `enums`, [9](#)
- `DEC_PRECISION`
 - `localsearch.h`, [95](#)
- `DataTable`
 - `mdata::DataTable`, [22](#)
- `dejong`
 - `mfunc::Functions`, [38](#)
- `eggHolder`
 - `mfunc::Functions`, [38](#)
- `enqueue`
 - `ThreadPool`, [81](#)
- `entryExists`
 - `util::IniReader`, [52](#)
- `enums`, [9](#)
 - `Algorithm`, [9](#)
 - `Count`, [9](#)
- `enums::AlgorithmNames`, [17](#)
 - `BLIND_SEARCH`, [18](#)
 - `get`, [17](#)
 - `LOCAL_SEARCH`, [18](#)
- `err`
 - `mdata::TestResult`, [78](#)

- exec
 - mfunc::Functions, 39
- execTime
 - mdata::TestResult, 79
- execTimesCol
 - mdata::TestParameters, 76
- execTimesRow
 - mdata::TestParameters, 76
- execTimesTable
 - mdata::TestParameters, 76
- Experiment
 - mfunc::Experiment, 27
- experiment.cpp
 - INI_FUNC_RANGE_SECTION, 128
 - INI_TEST_ALGORITHM, 128
 - INI_TEST_ALPHA, 128
 - INI_TEST_DIMENSIONS, 129
 - INI_TEST_EXECTIMESFILE, 129
 - INI_TEST_ITERATIONS, 129
 - INI_TEST_NUMTHREADS, 129
 - INI_TEST_POPULATION, 129
 - INI_TEST_RESULTSFILE, 130
 - INI_TEST_SECTION, 130
- exportCSV
 - mdata::DataTable, 22
- fitness
 - mdata::TestResult, 79
- funcld
 - mdata::TestParameters, 76
- generate
 - mdata::Population, 61
- get
 - enums::AlgorithmNames, 17
 - mfunc::FunctionDesc, 34
 - mfunc::Functions, 40
- getAllFitness
 - mdata::Population, 62
- getBestFitnessIndex
 - mdata::Population, 62
- getBestFitnessPtr
 - mdata::Population, 63
- getColLabel
 - mdata::DataTable, 24
- getDimensionsSize
 - mdata::Population, 63
- getEntry
 - mdata::DataTable, 25
 - util::IniReader, 53
- getEntryAs
 - util::IniReader, 53
- getFitness
 - mdata::Population, 65
- getFitnessPtr
 - mdata::Population, 65
- getPopulationPtr
 - mdata::Population, 66
- getPopulationSize
 - mdata::Population, 67
- griewangk
 - mfunc::Functions, 41
- INI_FUNC_RANGE_SECTION
 - experiment.cpp, 128
- INI_TEST_ALGORITHM
 - experiment.cpp, 128
- INI_TEST_ALPHA
 - experiment.cpp, 128
- INI_TEST_DIMENSIONS
 - experiment.cpp, 129
- INI_TEST_EXECTIMESFILE
 - experiment.cpp, 129
- INI_TEST_ITERATIONS
 - experiment.cpp, 129
- INI_TEST_NUMTHREADS
 - experiment.cpp, 129
- INI_TEST_POPULATION
 - experiment.cpp, 129
- INI_TEST_RESULTSFILE
 - experiment.cpp, 130
- INI_TEST_SECTION
 - experiment.cpp, 130
- include/blindsearch.h, 83, 85
- include/datatable.h, 85, 87
- include/experiment.h, 88, 90
- include/inireader.h, 91, 93
- include/localsearch.h, 93, 95
- include/mem.h, 97, 99
- include/mfuncptr.h, 100, 102
- include/mfunctions.h, 102, 109
- include/population.h, 114, 116
- include/searchalg.h, 117, 119
- include/stringutils.h, 120, 121
- include/testparam.h, 122, 123
- include/testresult.h, 124, 125
- include/threadpool.h, 125, 126
- IniReader
 - util::IniReader, 52
- init
 - mfunc::Experiment, 28
- initArray
 - util, 14
- initMatrix
 - util, 14
- isReady
 - mdata::Population, 67
- LOCAL_SEARCH
 - enums::AlgorithmNames, 18
- levy
 - mfunc::Functions, 41
- localsearch.h
 - DEC_PRECISION, 95
- main
 - main.cpp, 140
- main.cpp

- main, 140
- runExp, 140
- mastersCosineWave
 - mfunc::Functions, 42
- max
 - mfunc::RandomBounds, 70
- mdata, 10
- mdata::BlindSearch
 - run, 20
- mdata::BlindSearch< T >, 19
- mdata::DataTable
 - ~DataTable, 22
 - DataTable, 22
 - exportCSV, 22
 - getColLabel, 24
 - getEntry, 25
 - setColLabel, 25
 - setEntry, 26
- mdata::DataTable< T >, 21
- mdata::LocalSearch
 - run, 56
- mdata::LocalSearch< T >, 55
- mdata::Population
 - ~Population, 60
 - calcFitness, 60
 - generate, 61
 - getAllFitness, 62
 - getBestFitnessIndex, 62
 - getBestFitnessPtr, 63
 - getDimensionsSize, 63
 - getFitness, 65
 - getFitnessPtr, 65
 - getPopulationPtr, 66
 - getPopulationSize, 67
 - isReady, 67
 - outputFitness, 68
 - outputPopulation, 68
 - Population, 59
 - setFitness, 69
- mdata::Population< T >, 58
- mdata::SearchAlgorithm
 - ~SearchAlgorithm, 72
 - run, 72
 - SearchAlgorithm, 72
 - startTimer, 73
 - stopTimer, 73
 - timeDiff, 73
 - timer, 74
- mdata::SearchAlgorithm< T >, 71
- mdata::TestParameters
 - alg, 75
 - alpha, 75
 - execTimesCol, 76
 - execTimesRow, 76
 - execTimesTable, 76
 - funcId, 76
 - resultsCol, 77
 - resultsRow, 77
 - resultsTable, 77
 - TestParameters, 75
- mdata::TestParameters< T >, 74
- mdata::TestResult
 - err, 78
 - execTime, 79
 - fitness, 79
 - TestResult, 78
- mdata::TestResult< T >, 78
- mfunc, 10
 - mfuncPtr, 11
 - NUM_FUNCTIONS, 11
- mfunc::Experiment
 - ~Experiment, 27
 - Experiment, 27
 - init, 28
 - testAllFunc, 30
 - testFuncThreaded, 32
- mfunc::Experiment< T >, 26
- mfunc::FunctionDesc, 33
 - get, 34
- mfunc::Functions
 - ackleysOne, 36
 - ackleysTwo, 37
 - alpine, 37
 - dejong, 38
 - eggHolder, 38
 - exec, 39
 - get, 40
 - griewangk, 41
 - levy, 41
 - mastersCosineWave, 42
 - michalewicz, 43
 - nthroot, 43
 - pathological, 45
 - quartic, 45
 - rana, 46
 - rastrigin, 47
 - rosenbrok, 47
 - schwefel, 48
 - sineEnvelopeSineWave, 48
 - step, 49
 - stretchedVSineWave, 50
 - w, 50
- mfunc::Functions< T >, 35
- mfunc::RandomBounds
 - max, 70
 - min, 70
- mfunc::RandomBounds< T >, 70
- mfuncPtr
 - mfunc, 11
- mfunctions.h
 - _USE_MATH_DEFINES, 108
 - _ackleysOneDesc, 105
 - _ackleysTwoDesc, 105
 - _alpineDesc, 105
 - _dejongDesc, 105
 - _eggHolderDesc, 105

- [_griewangkDesc, 105](#)
 - [_levyDesc, 106](#)
 - [_mastersCosineWaveDesc, 106](#)
 - [_michalewiczDesc, 106](#)
 - [_pathologicalDesc, 106](#)
 - [_quarticDesc, 106](#)
 - [_ranaDesc, 107](#)
 - [_rastriginDesc, 107](#)
 - [_rosenbrokDesc, 107](#)
 - [_schwefelDesc, 107](#)
 - [_sineEnvelopeSineWaveDesc, 107](#)
 - [_stepDesc, 108](#)
 - [_stretchedVSineWaveDesc, 108](#)
- michalewicz
 - [mfunc::Functions, 43](#)
- min
 - [mfunc::RandomBounds, 70](#)
- NUM_FUNCTIONS
 - [mfunc, 11](#)
- nthroot
 - [mfunc::Functions, 43](#)
- openFile
 - [util::IniReader, 54](#)
- outputFitness
 - [mdata::Population, 68](#)
- outputPopulation
 - [mdata::Population, 68](#)
- pathological
 - [mfunc::Functions, 45](#)
- Population
 - [mdata::Population, 59](#)
- quartic
 - [mfunc::Functions, 45](#)
- rana
 - [mfunc::Functions, 46](#)
- rastrigin
 - [mfunc::Functions, 47](#)
- releaseArray
 - [util, 15](#)
- releaseMatrix
 - [util, 16](#)
- resultsCol
 - [mdata::TestParameters, 77](#)
- resultsRow
 - [mdata::TestParameters, 77](#)
- resultsTable
 - [mdata::TestParameters, 77](#)
- rosenbrok
 - [mfunc::Functions, 47](#)
- run
 - [mdata::BlindSearch, 20](#)
 - [mdata::LocalSearch, 56](#)
 - [mdata::SearchAlgorithm, 72](#)
- runExp
 - [main.cpp, 140](#)
- schwefel
 - [mfunc::Functions, 48](#)
- SearchAlgorithm
 - [mdata::SearchAlgorithm, 72](#)
- sectionExists
 - [util::IniReader, 54](#)
- setColLabel
 - [mdata::DataTable, 25](#)
- setEntry
 - [mdata::DataTable, 26](#)
- setFitness
 - [mdata::Population, 69](#)
- sineEnvelopeSineWave
 - [mfunc::Functions, 48](#)
- src/experiment.cpp, [127, 130](#)
- src/inireader.cpp, [136, 137](#)
- src/main.cpp, [139, 141](#)
- src/population.cpp, [142, 143](#)
- startTimer
 - [mdata::SearchAlgorithm, 73](#)
- step
 - [mfunc::Functions, 49](#)
- stopAndJoinAll
 - [ThreadPool, 81](#)
- stopTimer
 - [mdata::SearchAlgorithm, 73](#)
- stretchedVSineWave
 - [mfunc::Functions, 50](#)
- testAllFunc
 - [mfunc::Experiment, 30](#)
- testFuncThreaded
 - [mfunc::Experiment, 32](#)
- TestParameters
 - [mdata::TestParameters, 75](#)
- TestResult
 - [mdata::TestResult, 78](#)
- ThreadPool, [79](#)
 - [~ThreadPool, 80](#)
 - [enqueue, 81](#)
 - [stopAndJoinAll, 81](#)
 - [ThreadPool, 80](#)
- timeDiff
 - [mdata::SearchAlgorithm, 73](#)
- timer
 - [mdata::SearchAlgorithm, 74](#)
- util, [11](#)
 - [allocArray, 12](#)
 - [allocMatrix, 13](#)
 - [copyArray, 13](#)
 - [initArray, 14](#)
 - [initMatrix, 14](#)
 - [releaseArray, 15](#)
 - [releaseMatrix, 16](#)
- util::IniReader, [51](#)
 - [~IniReader, 52](#)

[entryExists](#), 52
[getEntry](#), 53
[getEntryAs](#), 53
[IniReader](#), 52
[openFile](#), 54
[sectionExists](#), 54

w

[mfunc::Functions](#), 50