CS471 Project 3

Generated by Doxygen 1.8.13

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 mdata Namespace Reference

Classes

- class DataTable

  The DataTable class is a simple table of values with labeled columns.
- class Population

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

4.2 mfunc Namespace Reference

Classes

- class Experiment

  Contains classes for running the CS471 project experiment.
- struct FAParams

  The FAParams struct contains various parameters that are required to be passed to the Firefly.run() method.
- class Firefly

  The Firefly class runs the firefly algorithm with the given parameters passed to the run() method.
- 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.
- class HarmonySearch

  The HarmonySearch class runs the harmony search algorithm based on the parameters passed to the run() method.
- struct HSParams

  The HSParams struct contains various parameters that are required to be passed to the HarmonySearch.run() method.
- struct Particle

  The Particle struct is a simple data structure used to store the global best particle along with it's fitness.
- class ParticleSwarm

  The ParticleSwarm class runs the particle swarm algorithm with the given parameters passed to the run() method.
- struct PSParams

  The PSParams struct contains various parameters that are required to be passed to the ParticleSwarm.run() method.
- 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.

Enumerations

- enum Algorithm { Algorithm::ParticleSwarm = 0, Algorithm::Firefly = 1, Algorithm::HarmonySearch = 2, Algorithm::Count = 3 }

  Simple enum that selects one of the search algorithms.

Variables

- constexpr const unsigned int NUM_FUNCTIONS = _NUM_FUNCTIONS

4.2.1 Detailed Description

Scope for all math functions

4.2.2 Typedef Documentation

4.2.2.1 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

T	Data type for vector and return value

Definition at line 28 of file mfuncptr.h.

4.2.3 Enumeration Type Documentation

4.2.3.1 Algorithm

```
enum mfunc::Algorithm  [strong]
```

Simple enum that selects one of the search algorithms.

Enumerator

ParticleSwarm	
Firefly	
HarmonySearch	
Count	

Definition at line 44 of file experiment.h.

```
00045     {
00046         ParticleSwarm = 0,
00047         Firefly = 1,
00048         HarmonySearch = 2,
00049         Count = 3
00050     };
```

4.2.4 Variable Documentation

4.2.4.1 NUM_FUNCTIONS

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

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

Definition at line 67 of file mfunctions.h.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

4.3 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>
  bool 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.

4.3.1 Function Documentation

4.3.1.1 allocArray()

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

Allocates a new array of the given data type.

Template Parameters

Data	type of the array

Parameters

size	Number of elements in the array

Returns

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

Definition at line 116 of file mem.h.

```
00117     {
00118         return new(std::nothrow) T[size];
00119     }
```

4.3.1.2 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

Data	type of the matrix entries

Parameters

rows	The number of rows
cols	The number of columns

Returns

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

Definition at line 130 of file mem.h.

```
00131     {
00132         T** m = (T**)allocArray<T*>(rows);
00133         if (m == nullptr) return nullptr;
00134 
00135         for (size_t i = 0; i < rows; i++)
00136         {
00137             m[i] = allocArray<T>(cols);
00138             if (m[i] == nullptr)
00139             {
00140                 releaseMatrix<T>(m, rows);
00141                 return nullptr;
00142             }
00143         }
00144 
00145         return m;
00146     }
```

4.3.1.3 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

Data	type of the array

Parameters

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

Definition at line 157 of file mem.h.

```
00158     {
00159         for (size_t i = 0; i < size; i++)
00160             dest[i] = src[i];
00161     }
```

4.3.1.4 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

Data	type of array

Parameters

a	Pointer to array
size	Size of the array
val	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     }
```

4.3.1.5 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

m	Pointer to a matrix
rows	Number of rows in matrix
cols	Number of columns in matrix
val	Value to initialize the matrix to

Definition at line 49 of file mem.h.

References initArray().

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

4.3.1.6 releaseArray()

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

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

Template Parameters

Data	type of array

Parameters

a	Pointer to array

Definition at line 66 of file mem.h.

Referenced by mfunc::Firefly< T >::run().

```
00067     {
00068         if (a == nullptr) return true;
00069 
00070         try
00071         {
00072             delete[] a;
00073             a = nullptr;
00074             return true;
00075         }
00076         catch(...)
00077         {
00078             return false;
00079         }
00080     }
```

4.3.1.7 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

Data	type of the matrix

Parameters

m	Pointer th the matrix
rows	The number of rows in the matrix

Definition at line 90 of file mem.h.

Referenced by mdata::DataTable< T >::~DataTable().

```
00091     {
00092         if (m == nullptr) return;
00093 
00094         for (size_t i = 0; i < rows; i++)
00095         {
00096             if (m[i] != nullptr)
00097             {
00098                 // Release each row
00099                 releaseArray<T>(m[i]);
00100             }
00101         }
00102 
00103         // Release columns
00104         delete[] m;
00105         m = nullptr;
00106     }
```

Chapter 5

Class Documentation

5.1 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.
- void clearData ()
- 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.

5.1.1 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.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 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

_rows	Number of rows in table
_cols	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         }
```

5.1.2.2 ~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         }
```

5.1.3 Member Function Documentation

5.1.3.1 clearData()

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

Definition at line 82 of file datatable.h.

```
00083         {
00084             util::initMatrix<T>(dataMatrix, rows, cols, 0);
00085         }
```

5.1.3.2 exportCSV()

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

Exports the contents of this DataTable to a .csv file.

Parameters

filePath	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 160 of file datatable.h.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00161         {
00162             if (dataMatrix == nullptr) return false;
00163 
00164             using namespace std;
00165             ofstream outFile;
00166             outFile.open(filePath, ofstream::out | ofstream::trunc);
00167             if (!outFile.good()) return false;
00168 
00169             // Print column labels
00170             for (unsigned int c = 0; c < cols; c++)
00171             {
00172                 outFile << colLabels[c];
00173                 if (c < cols - 1) outFile << ",";
00174             }
00175 
00176             outFile << endl;
00177 
00178             // Print data rows
00179             for (unsigned int r = 0; r < rows; r++)
00180             {
00181                 for (unsigned int c = 0; c < cols; c++)
00182                 {
00183                     outFile << std::setprecision(8) << dataMatrix[r][c];
00184                     if (c < cols - 1) outFile << ",";
00185                 }
00186                 outFile << endl;
00187             }
00188 
00189             outFile.close();
00190             return true;
00191         }
```

5.1.3.3 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

colIndex	Index of the column

Returns

std::string String value of the column label

Definition at line 93 of file datatable.h.

```
00094         {
00095             if (colIndex >= colLabels.size())
00096                 throw std::out_of_range("Column index out of range");
00097 
00098             return colLabels[colIndex];
00099         }
```

5.1.3.4 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

row	Row index of the table
col	Column index of the table

Returns

T Value of the entry at the given row and column

Definition at line 122 of file datatable.h.

```
00123         {
00124             if (dataMatrix == nullptr)
00125                 throw std::runtime_error("Data matrix not allocated");
00126             if (row >= rows)
00127                 throw std::out_of_range("Table row out of range");
00128             else if (col >= cols)
00129                 throw std::out_of_range("Table column out of range");
00130 
00131             return dataMatrix[row][col];
00132         }
```

5.1.3.5 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

colIndex	Index of the column
newLabel	New string label for the column

Definition at line 107 of file datatable.h.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00108         {
```

```
00109             if (colIndex >= colLabels.size())
00110                throw std::out_of_range("Column index out of range");
00111
00112             colLabels[colIndex] = newLabel;
00113         }
```

5.1.3.6 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

row	Row index of the table
col	Column index of the table
val	New value for the entry

Definition at line 141 of file datatable.h.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00142         {
00143             if (dataMatrix == nullptr)
00144                 throw std::runtime_error("Data matrix not allocated");
00145             if (row >= rows)
00146                 throw std::out_of_range("Table row out of range");
00147             else if (col >= cols)
00148                 throw std::out_of_range("Table column out of range");
00149
00150             dataMatrix[row][col] = val;
00151         }
```

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

- include/datatable.h

5.2 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 4 document, records results, and outputs the data as a *.csv file.*
- int testPS ()

  Tests the particle swarm algorithm for all 18 functions and then outputs the results files.
- int testFA ()

  Tests the firefly algorithm for all 18 functions and then outputs the results files.
- int testHS ()

  Tests the harmony search algorithm for all 18 functions and then outputs the results files.

5.2.1 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 63 of file experiment.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Experiment()

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

Construct a new Experiment object.

Definition at line 75 of file experiment.cpp.

```
00076     : vBounds(nullptr), tPool(nullptr), resultsFile(""), execTimesFile(""), iterations(0)
00077 {
00078 }
```

5.2.2.2 ~Experiment()

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

Destroys the Experiment object.

Definition at line 85 of file experiment.cpp.

```
00086 {
00087     releaseThreadPool();
00088     releasePopulationPool();
00089     releaseVBounds();
00090 }
```

5.2.3 Member Function Documentation

5.2.3.1 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

paramFile	File path to the parameter ini file

Returns

Returns true if initialization was successful. Otherwise false.

Definition at line 101 of file experiment.cpp.

References mfunc::Count, util::IniReader::getEntry(), util::IniReader::getEntryAs(), INI_TEST_ALGORITHM, INI_TEST_DIMENSIONS, INI_TEST_EXECTIMESFILE, INI_TEST_FUNCCALLSFILE, INI_TEST_ITERATIONS, INI_TEST_NUMTHREADS, INI_TEST_POPULATION, INI_TEST_POPULATIONFILE, INI_TEST_RESULTSFILE, INI_TEST_SECTION, INI_TEST_WORSTFITNESSFILE, and util::IniReader::openFile().

Referenced by runExp().

```
00102 {
00103     try
00104     {
00105         // Open and parse parameters file
00106         if (!iniParams.openFile(paramFile))
00107         {
00108             cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
00109             return false;
00110         }
00111
```

```
00112         // Extract test parameters from ini file
00113         long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_POPULATION);
00114         long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_DIMENSIONS);
00115         long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_ITERATIONS);
00116         long numberThreads = iniParams.getEntryAs<long>(
      INI_TEST_SECTION, INI_TEST_NUMTHREADS);
00117         unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(
      INI_TEST_SECTION, INI_TEST_ALGORITHM);
00118         resultsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_RESULTSFILE);
00119         worstFitnessFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_WORSTFITNESSFILE);
00120         execTimesFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_EXECTIMESFILE);
00121         funcCallsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_FUNCCALLSFILE);
00122         populationsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_POPULATIONFILE);
00123
00124         // Verify test parameters
00125         if (numberSol <= 0)
00126         {
00127             cerr << "Experiment init failed: Param file [test]->"
00128                 << INI_TEST_POPULATION << " entry missing or out of bounds: " <<
      paramFile << endl;
00129             return false;
00130         }
00131         else if (numberDim <= 0)
00132         {
00133             cerr << "Experiment init failed: Param file [test]->"
00134                 << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " <<
      paramFile << endl;
00135             return false;
00136         }
00137         else if (numberIter <= 0)
00138         {
00139             cerr << "Experiment init failed: Param file [test]->"
00140                 << INI_TEST_ITERATIONS << " entry missing or out of bounds: " <<
      paramFile << endl;
00141             return false;
00142         }
00143         else if (numberThreads <= 0)
00144         {
00145             cerr << "Experiment init failed: Param file [test]->"
00146                 << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " <<
      paramFile << endl;
00147             return false;
00148         }
00149         else if (selectedAlg >= static_cast<unsigned int>(Algorithm::Count))
00150         {
00151             cerr << "Experiment init failed: Param file [test]->"
00152                 << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile
       << endl;
00153             return false;
00154         }
00155
00156         // Cast iterations and test algorithm to correct types
00157         iterations = (size_t)numberIter;
00158         selAlg = static_cast<Algorithm>(selectedAlg);
00159
00160         // Print test parameters to console
00161         cout << "Population size: " << numberSol << endl;
00162         cout << "Dimensions: " << numberDim << endl;
00163         cout << "Iterations: " << iterations << endl;
00164
00165         // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
00166         if (!allocatePopulationPool((size_t)numberThreads * 2, (size_t)numberSol, (size_t)numberDim))
00167         {
00168             cerr << "Experiment init failed: Unable to allocate populations." << endl;
00169             return false;
00170         }
00171
00172         // Allocate memory for function vector bounds
00173         if (!allocateVBounds())
00174         {
00175             cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
00176             return false;
00177         }
00178
00179         // Fill function bounds array with data parsed from iniParams
00180         if (!parseFuncBounds())
00181         {
00182             cerr << "Experiment init failed: Unable to parse vector bounds array." << endl;
00183             return false;
```

```
00184         }
00185
00186         // Allocate thread pool
00187         if (!allocateThreadPool((size_t)numberThreads))
00188         {
00189             cerr << "Experiment init failed: Unable to allocate thread pool." << endl;
00190             return false;
00191         }
00192
00193         cout << "Started " << numberThreads << " worker threads ..." << endl;
00194
00195         // Ready to run an experiment
00196         return true;
00197     }
00198     catch (const std::exception& ex)
00199     {
00200         cerr << "Exception occurred while initializing experiment: " << ex.what() << endl;
00201         return false;
00202     }
00203     catch (...)
00204     {
00205         cerr << "Unknown Exception occurred while initializing experiment." << endl;
00206         return false;
00207     }
00208 }
```

5.2.3.2 testAllFunc()

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

Executes all functions as specified in the CS471 project 4 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 217 of file experiment.cpp.

References mfunc::Firefly, mfunc::HarmonySearch, mfunc::ParticleSwarm, mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

Referenced by runExp().

```
00218 {
00219     // Run the selected algorithm
00220     switch (selAlg)
00221     {
00222     case Algorithm::ParticleSwarm:
00223         return testPS();
00224         break;
00225     case Algorithm::Firefly:
00226         return testFA();
00227         break;
00228     case Algorithm::HarmonySearch:
00229         return testHS();
00230         break;
00231     default:
00232         cout << "Error: Invalid algorithm selected." << endl;
00233         break;
00234     }
00235
00236     return 1;
00237 }
```

5.2.3.3 testFA()

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

Tests the firefly algorithm for all 18 functions and then outputs the results files.

Returns

Returns a non-zero error code on failure, otherwise returns zero on success

Definition at line 387 of file experiment.cpp.

References mfunc::FAParams< T >::bestFitnessTable, ThreadPool::enqueue(), mdata::DataTable< T >::exportCSV(), mfunc::FAParams< T >::fitTableCol, mfunc::FAParams< T >::fMaxBound, mfunc::FAParams< T >::fMinBound, mfunc::FAParams< T >::fPtr, mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), mfunc::FAParams< T >::iterations, mfunc::FAParams< T >::mainPop, mfunc::FAParams< T >::nextPop, mfunc::NUM_FUNCTIONS, mfunc::FAParams< T >::popFile, mfunc::Functions< T >::resetCallCounters(), RESULTSFILE_ALG_PATTERN, mfunc::Firefly< T >::run(), mdata::DataTable< T >::setColLabel(), mdata::DataTable< T >::setEntry(), and mfunc::FAParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testAllFunc().

```
00388 {
00389     // Prepare alg parameter template struct and results tables
00390     const FAParams<T> paramTemplate = createFAParamsTemplate();
00391     mdata::DataTable<T> resultsTable(iterations, 18);
00392     mdata::DataTable<T> worstTable(iterations, 18);
00393     mdata::DataTable<T> execTimesTable(1, 18);
00394     mdata::DataTable<T> funcCallsTable(1, 18);
00395     std::vector<std::future<int>> testFutures;
00396 
00397     // Reset objective function call counters
00398     mfunc::Functions<T>::resetCallCounters();
00399 
00400     // Queue up a threaded task for each of the 18 objective functions
00401     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00402     {
00403         // Set results table column labels
00404         auto desc = mfunc::FunctionDesc::get(f);
00405         resultsTable.setColLabel(f - 1, desc);
00406         worstTable.setColLabel(f - 1, desc);
00407         execTimesTable.setColLabel(f - 1, desc);
00408         funcCallsTable.setColLabel(f - 1, desc);
00409 
00410         // Create new parameters struct for current function and set parameters
00411         FAParams<T> params(paramTemplate);
00412         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00413         params.bestFitnessTable = &resultsTable;
00414         params.worstFitnessTable = &worstTable;
00415         params.fitTableCol = f - 1;
00416         params.mainPop = nullptr;
00417         params.fPtr = mfunc::Functions<T>::get(f);
00418         params.fMinBound = vBounds[f-1].min;
00419         params.fMaxBound = vBounds[f-1].max;
00420         params.iterations = iterations;
00421 
00422         // Add search algorithm run to thread pool queue
00423         testFutures.emplace_back(
00424                 tPool->enqueue(&Experiment<T>::runFAThreaded, this,
      params, &execTimesTable, 0, f - 1)
00425         );
00426     }
00427 
00428     cout << "Executing firefly ..." << endl << flush;
00429 
00430     // Wait for all threads to finish
00431     waitThreadFutures(testFutures);
00432     testFutures.clear();
00433 
00434     cout << endl;
00435 
00436     // Output objective function call counter values to .csv file
```

```
00437     if (!funcCallsFile.empty())
00438     {
00439         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00440             funcCallsTable.setEntry(0, f - 1,
      mfunc::Functions<T>::getCallCounter(f));
00441
00442         std::string outFile = util::s_replace(funcCallsFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00443         if (funcCallsTable.exportCSV(outFile.c_str()))
00444             cout << "Function call counts written to: " << outFile << endl;
00445         else
00446             cout << "Unable to function call counts file: " << outFile << endl;
00447     }
00448
00449     // Output best fitness values to .csv file
00450     if (!resultsFile.empty())
00451     {
00452         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00453         if (resultsTable.exportCSV(outFile.c_str()))
00454             cout << "Best fitness results written to: " << outFile << endl;
00455         else
00456             cout << "Unable to open results file: " << outFile << endl;
00457     }
00458
00459     // Output worst fitness values to .csv file
00460     if (!worstFitnessFile.empty())
00461     {
00462         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00463         if (worstTable.exportCSV(outFile.c_str()))
00464             cout << "Worst fitness results written to: " << outFile << endl;
00465         else
00466             cout << "Unable to open worst fitness file: " << outFile << endl;
00467     }
00468
00469     // Output execution times to .csv file
00470     if (!execTimesFile.empty())
00471     {
00472         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00473         if (execTimesTable.exportCSV(outFile.c_str()))
00474             cout << "Execution times written to: " << outFile << endl;
00475         else
00476             cout << "Unable to open execution times file: " << outFile << endl;
00477     }
00478
00479     return 0;
00480 }
```

5.2.3.4 testHS()

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

Tests the harmony search algorithm for all 18 functions and then outputs the results files.

Returns

Returns a non-zero error code on failure, otherwise returns zero on success

Definition at line 527 of file experiment.cpp.

References mfunc::FAParams< T >::alpha, mfunc::HSParams< T >::bestFitnessTable, mfunc::FAParams< T >::betamin, mfunc::HSParams< T >::bw, mfunc::PSParams< T >::c1, mfunc::PSParams< T >::c2, ThreadPool::enqueue(), mdata::DataTable< T >::exportCSV(), mfunc::HSParams< T >::fitTableCol, mfunc::HSParams<

T >::fMaxBound, mfunc::HSParams< T >::fMinBound, mfunc::HSParams< T >::fPtr, mfunc::FAParams< T >::gamma, mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), util::IniReader::getEntry(), util::IniReader::getEntryAs(), mfunc::HSParams< T >::hmcr, INI_FA_ALPHA, INI_FA_BETAMIN, INI_FA_GAMMA, INI_FA_SECTION, INI_HS_BW, INI_HS_HMCR, INI_HS_PAR, INI_HS_SECTION, INI_PSO_C1, INI_PSO_C2, INI_PSO_K, INI_PSO_SECTION, mfunc::HSParams< T >::iterations, mfunc::PSParams< T >::k, mfunc::HSParams< T >::mainPop, mfunc::RandomBounds< T >::max, mfunc::RandomBounds< T >::min, mfunc::NUM_FUNCTIONS, mfunc::HSParams< T >::par, PARAM_DEFAULT_FA_ALPHA, PARAM_DEFAULT_FA_BETAMIN, PARAM_DEFAULT_FA_GAMMA, PARAM_DEFAULT_HS_BW, PARAM_DEFAULT_HS_HMCR, PARAM_DEFAULT_HS_PAR, PARAM_DEFAULT_PSO_C1, PARAM_DEFAULT_PSO_C2, PARAM_DEFAULT_PSO_K, mfunc::HSParams< T >::popFile, mfunc::Functions< T >::resetCallCounters(), RESULTSFILE_ALG_PATTERN, mfunc::HarmonySearch< T >::run(), mdata::DataTable< T >::setColLabel(), mdata::DataTable< T >::setEntry(), ThreadPool::stopAndJoinAll(), and mfunc::HSParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testAllFunc().

```
00528 {
00529     // Prepare alg parameter template struct and results tables
00530     const HSParams<T> paramTemplate = createHSParamsTemplate();
00531     mdata::DataTable<T> resultsTable(iterations, 18);
00532     mdata::DataTable<T> worstTable(iterations, 18);
00533     mdata::DataTable<T> execTimesTable(1, 18);
00534     mdata::DataTable<T> funcCallsTable(1, 18);
00535     std::vector<std::future<int>> testFutures;
00536 
00537     // Reset objective function call counters
00538     mfunc::Functions<T>::resetCallCounters();
00539 
00540     // Queue up a threaded task for each of the 18 objective functions
00541     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00542     {
00543         // Set results table column labels
00544         auto desc = mfunc::FunctionDesc::get(f);
00545         resultsTable.setColLabel(f - 1, desc);
00546         worstTable.setColLabel(f - 1, desc);
00547         execTimesTable.setColLabel(f - 1, desc);
00548         funcCallsTable.setColLabel(f - 1, desc);
00549 
00550         // Create new parameters struct for current function and set parameters
00551         HSParams<T> params(paramTemplate);
00552         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00553         params.bestFitnessTable = &resultsTable;
00554         params.worstFitnessTable = &worstTable;
00555         params.fitTableCol = f - 1;
00556         params.mainPop = nullptr;
00557         params.fPtr = mfunc::Functions<T>::get(f);
00558         params.fMinBound = vBounds[f-1].min;
00559         params.fMaxBound = vBounds[f-1].max;
00560         params.iterations = iterations;
00561 
00562         // Add search algorithm run to thread pool queue
00563         testFutures.emplace_back(
00564                 tPool->enqueue(&Experiment<T>::runHSThreaded, this,
    params, &execTimesTable, 0, f - 1)
00565         );
00566     }
00567 
00568     cout << "Executing harmony search ..." << endl << flush;
00569 
00570     waitThreadFutures(testFutures);
00571 
00572     // Clear thread futures
00573     testFutures.clear();
00574 
00575     cout << endl;
00576 
00577     // Output objective function call counter values to .csv file
00578     if (!funcCallsFile.empty())
00579     {
00580         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00581             funcCallsTable.setEntry(0, f - 1,
    mfunc::Functions<T>::getCallCounter(f));
00582 
00583         std::string outFile = util::s_replace(funcCallsFile,
    RESULTSFILE_ALG_PATTERN, "HS");
00584         if (funcCallsTable.exportCSV(outFile.c_str()))
00585             cout << "Function call counts written to: " << outFile << endl;
00586         else
00587             cout << "Unable to function call counts file: " << outFile << endl;
00588     }
```

```
00589
00590     // Output best fitness values to .csv file
00591     if (!resultsFile.empty())
00592     {
00593         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00594         if (resultsTable.exportCSV(outFile.c_str()))
00595             cout << "Best fitness results written to: " << outFile << endl;
00596         else
00597             cout << "Unable to open results file: " << outFile << endl;
00598     }
00599
00600     // Output worst fitness values to .csv file
00601     if (!worstFitnessFile.empty())
00602     {
00603         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00604         if (worstTable.exportCSV(outFile.c_str()))
00605             cout << "Worst fitness results written to: " << outFile << endl;
00606         else
00607             cout << "Unable to open worst fitness file: " << outFile << endl;
00608     }
00609
00610     // Output execution times to .csv file
00611     if (!execTimesFile.empty())
00612     {
00613         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00614         if (execTimesTable.exportCSV(outFile.c_str()))
00615             cout << "Execution times written to: " << outFile << endl;
00616         else
00617             cout << "Unable to open execution times file: " << outFile << endl;
00618     }
00619
00620     return 0;
00621 }
```

5.2.3.5 testPS()

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

Tests the particle swarm algorithm for all 18 functions and then outputs the results files.

Returns

Returns a non-zero error code on failure, otherwise returns zero on success

Definition at line 246 of file experiment.cpp.

References mfunc::PSParams< T >::bestFitnessTable, ThreadPool::enqueue(), mdata::DataTable< T >::exportCSV(), mfunc::PSParams< T >::fitTableCol, mfunc::PSParams< T >::fMaxBound, mfunc::PSParams< T >::fMinBound, mfunc::PSParams< T >::fPtr, mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), mfunc::PSParams< T >::iterations, mfunc::PSParams< T >::mainPop, mfunc::NUM_FUNCTIONS, mfunc::PSParams< T >::pbPop, mfunc::PSParams< T >::popFile, mfunc::Functions< T >::resetCallCounters(), RESULTSFILE_ALG_PATTERN, mfunc::ParticleSwarm< T >::run(), mdata::DataTable< T >::setColLabel(), mdata::DataTable< T >::setEntry(), and mfunc::PSParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testAllFunc().

```
00247 {
00248     // Prepare alg parameter template struct and results tables
00249     const PSParams<T> paramTemplate = createPSParamsTemplate();
00250     mdata::DataTable<T> resultsTable(iterations, 18);
00251     mdata::DataTable<T> worstTable(iterations, 18);
00252     mdata::DataTable<T> execTimesTable(1, 18);
00253     mdata::DataTable<T> funcCallsTable(1, 18);
00254     std::vector<std::future<int>> testFutures;
00255 
00256     // Reset objective function call counters
00257     mfunc::Functions<T>::resetCallCounters();
00258 
00259     // Queue up a threaded task for each of the 18 objective functions
00260     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00261     {
00262         // Set results table column labels
00263         auto desc = mfunc::FunctionDesc::get(f);
00264         resultsTable.setColLabel(f - 1, desc);
00265         worstTable.setColLabel(f - 1, desc);
00266         execTimesTable.setColLabel(f - 1, desc);
00267         funcCallsTable.setColLabel(f - 1, desc);
00268 
00269         // Create new parameters struct for current function and set parameters
00270         PSParams<T> params(paramTemplate);
00271         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00272         params.bestFitnessTable = &resultsTable;
00273         params.worstFitnessTable = &worstTable;
00274         params.fitTableCol = f - 1;
00275         params.mainPop = nullptr;
00276         params.pbPop = nullptr;
00277         params.fPtr = mfunc::Functions<T>::get(f);
00278         params.fMinBound = vBounds[f-1].min;
00279         params.fMaxBound = vBounds[f-1].max;
00280         params.iterations = iterations;
00281 
00282         // Add search algorithm run to thread pool queue
00283         testFutures.emplace_back(
00284                 tPool->enqueue(&Experiment<T>::runPSThreaded, this,
      params, &execTimesTable, 0, f - 1)
00285         );
00286     }
00287 
00288     cout << "Executing particle swarm ..." << endl << flush;
00289 
00290     // Wait for threads to finish running all functions
00291     waitThreadFutures(testFutures);
00292     testFutures.clear();
00293 
00294     cout << endl;
00295 
00296     // Output objective function call counter values to .csv file
00297     if (!funcCallsFile.empty())
00298     {
00299         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00300             funcCallsTable.setEntry(0, f - 1,
      mfunc::Functions<T>::getCallCounter(f));
00301 
00302         std::string outFile = util::s_replace(funcCallsFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00303         if (funcCallsTable.exportCSV(outFile.c_str()))
00304             cout << "Function call counts written to: " << outFile << endl;
00305         else
00306             cout << "Unable to function call counts file: " << outFile << endl;
00307     }
00308 
00309     // Output best fitness values to .csv file
00310     if (!resultsFile.empty())
00311     {
00312         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00313         if (resultsTable.exportCSV(outFile.c_str()))
00314             cout << "Best fitness results written to: " << outFile << endl;
00315         else
00316             cout << "Unable to open results file: " << outFile << endl;
00317     }
00318 
00319     // Output worst fitness values to .csv file
00320     if (!worstFitnessFile.empty())
00321     {
00322         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00323         if (worstTable.exportCSV(outFile.c_str()))
00324             cout << "Worst fitness results written to: " << outFile << endl;
00325         else
00326             cout << "Unable to open worst fitness file: " << outFile << endl;
00327     }
00328 
```

```
00329     // Output execution times to .csv file
00330     if (!execTimesFile.empty())
00331     {
00332         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00333         if (execTimesTable.exportCSV(outFile.c_str()))
00334             cout << "Execution times written to: " << outFile << endl;
00335         else
00336             cout << "Unable to open execution times file: " << outFile << endl;
00337     }
00338
00339     return 0;
00340 }
```

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

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

5.3 mfunc::FAParams< T > Struct Template Reference

The FAParams struct contains various parameters that are required to be passed to the Firefly.run() method.

`#include <firefly.h>`

Public Member Functions

- FAParams ()

  Construct a new FAParams object.

Public Attributes

- std::string popFile
- mdata::DataTable< T > * bestFitnessTable
- mdata::DataTable< T > * worstFitnessTable
- size_t fitTableCol
- mdata::Population< T > * mainPop
- mdata::Population< T > * nextPop
- mfuncPtr< T > fPtr
- T fMinBound
- T fMaxBound
- unsigned int iterations
- double alpha
- double betamin
- double gamma

5.3.1 Detailed Description

template<class T>
struct mfunc::FAParams< T >

The FAParams struct contains various parameters that are required to be passed to the Firefly.run() method.

Template Parameters

T	Data type used by the search algorithm

Definition at line 39 of file firefly.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 FAParams()

```
template<class T>
mfunc::FAParams< T >::FAParams ( )  [inline]
```

Construct a new FAParams object.

Definition at line 58 of file firefly.h.

```
00059         {
00060             popFile = "";
00061             bestFitnessTable = nullptr;
00062             worstFitnessTable = nullptr;
00063             fitTableCol = 0;
00064             mainPop = nullptr;
00065             nextPop = nullptr;
00066             fPtr = nullptr;
00067             fMinBound = 0;
00068             fMaxBound = 0;
00069             iterations = 0;
00070             alpha = 0;
00071             betamin = 0;
00072             gamma = 0;
00073         }
```

5.3.3 Member Data Documentation

5.3.3.1 alpha

```
template<class T>
double mfunc::FAParams< T >::alpha
```

Definition at line 51 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testHS().

5.3.3.2 bestFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::FAParams< T >::bestFitnessTable
```

Definition at line 42 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.3 betamin

```
template<class T>
double mfunc::FAParams< T >::betamin
```

Definition at line 52 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testHS().

5.3.3.4 fitTableCol

```
template<class T>
size_t mfunc::FAParams< T >::fitTableCol
```

Definition at line 44 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.5 fMaxBound

```
template<class T>
T mfunc::FAParams< T >::fMaxBound
```

Definition at line 49 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.6 fMinBound

```
template<class T>
T mfunc::FAParams< T >::fMinBound
```

Definition at line 48 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.7 fPtr

```
template<class T>
mfuncPtr<T> mfunc::FAParams< T >::fPtr
```

Definition at line 47 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.8 gamma

```
template<class T>
double mfunc::FAParams< T >::gamma
```

Definition at line 53 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testHS().

5.3.3.9 iterations

```
template<class T>
unsigned int mfunc::FAParams< T >::iterations
```

Definition at line 50 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.10 mainPop

```
template<class T>
mdata::Population<T>* mfunc::FAParams< T >::mainPop
```

Definition at line 45 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.11 nextPop

```
template<class T>
mdata::Population<T>* mfunc::FAParams< T >::nextPop
```

Definition at line 46 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.12 popFile

```
template<class T>
std::string mfunc::FAParams< T >::popFile
```

Definition at line 41 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

5.3.3.13 worstFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::FAParams< T >::worstFitnessTable
```

Definition at line 43 of file firefly.h.

Referenced by mfunc::Firefly< T >::run(), and mfunc::Experiment< T >::testFA().

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

- include/firefly.h

5.4 mfunc::Firefly< T > Class Template Reference

The Firefly class runs the firefly algorithm with the given parameters passed to the run() method.

```
#include <firefly.h>
```

Public Member Functions

- Firefly ()
  Construct a new Firefly object.
- ~Firefly ()=default
- int run (FAParams< T > p)
  Runs the firefly algorithm with the given parameters.

5.4.1 Detailed Description

template<class T>
class mfunc::Firefly< T >

The Firefly class runs the firefly algorithm with the given parameters passed to the run() method.

Template Parameters

T	Data type used by the algorithm

Definition at line 83 of file firefly.h.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Firefly()

```
template<class T >
mfunc::Firefly< T >::Firefly ( )
```

Construct a new Firefly object.

Template Parameters

T	Data type used by the algorithm

Definition at line 106 of file firefly.h.

```
00107     : seed(), engine(seed()), rchance(0, 1)
00108 {
00109 }
```

5.4.2.2 ~Firefly()

```
template<class T>
mfunc::Firefly< T >::~Firefly ( )  [default]
```

5.4.3 Member Function Documentation

5.4.3.1 run()

```
template<class T >
int mfunc::Firefly< T >::run (
            FAParams< T > p )
```

Runs the firefly algorithm with the given parameters.

Template Parameters

T	Data type used by the algorithm

Parameters

p	Parameters for the algorithm

Returns

Returns a non-zero error code on failure, or zero on success

Definition at line 119 of file firefly.h.

References mfunc::FAParams< T >::alpha, mfunc::FAParams< T >::bestFitnessTable, BETA_INIT, mfunc::FAParams< T >::betamin, mfunc::FAParams< T >::fitTableCol, mfunc::FAParams< T >::fMaxBound, mfunc::FAParams< T >::fMinBound, mfunc::FAParams< T >::fPtr, mfunc::FAParams< T >::gamma, mfunc::FAParams< T >::iterations, mfunc::FAParams< T >::mainPop, mfunc::FAParams< T >::nextPop, mfunc::FAParams< T >::popFile, POPFILE_GEN_PATTERN, util::releaseArray(), and mfunc::FAParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testFA().

```
00120 {
00121     if (p.mainPop == nullptr || p.nextPop == nullptr || p.fPtr == nullptr)
00122         return 1;
00123 
00124     // Get population information
00125     const size_t popSize = p.mainPop->getPopulationSize();
00126     const size_t dimSize = p.mainPop->getDimensionsSize();
00127 
00128     T* solBuffer = util::allocArray<T>(dimSize);
00129     if (solBuffer == nullptr)
00130         return 2;
00131 
00132     // Generate population vectors
00133     if (!p.nextPop->generate(p.fMinBound, p.fMaxBound))
00134         return 3;
00135 
00136     // Calculate fitness for all population vectors
00137     if (!p.nextPop->calcAllFitness(p.fPtr))
00138         return 4;
00139 
00140     // Sort population from worst to best
00141     p.nextPop->sortFitnessDescend();
00142 
00143     for (unsigned int iter = 0; iter < p.iterations; iter++)
00144     {
00145         p.mainPop->copyAllFrom(p.nextPop);
00146 
00147         for (size_t firefly_i = 0; firefly_i < popSize; firefly_i++)
00148         {
00149             evaluate(p, solBuffer, firefly_i);
00150         }
00151 
00152         p.nextPop->sortFitnessDescend();
00153 
00154         // Store best fitness for this iteration
00155         if (p.bestFitnessTable != nullptr)
00156             p.bestFitnessTable->setEntry(iter, p.fitTableCol, p.nextPop->getFitness(popSize - 1));
00157 
00158         // Store worst fitness for this iteration
00159         if (p.worstFitnessTable != nullptr)
00160             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.nextPop->getFitness(0));
00161 
00162         // Dump population vectors to file
00163         if (!p.popFile.empty())
00164             p.nextPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
      POPFILE_GEN_PATTERN), std::to_string(iter)));
00165     }
00166 
00167     util::releaseArray(solBuffer);
00168 
00169     return 0;
00170 }
```

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

- include/firefly.h

5.5 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)

5.5.1 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

f	Function id to retrieve the description for

Returns

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

Definition at line 76 of file mfunctions.h.

5.5.2 Member Function Documentation

5.5.2.1 get()

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

Definition at line 78 of file mfunctions.h.

References _ackleysOneDesc, _ackleysOneId, _ackleysTwoDesc, _ackleysTwoId, _alpineDesc, _alpineId, _dejongDesc, _dejongId, _eggHolderDesc, _eggHolderId, _griewangkDesc, _griewangkId, _levyDesc, _levyId, _mastersCosineWaveDesc, _mastersCosineWaveId, _michalewiczDesc, _michalewiczId, _pathologicalDesc, _pathologicalId, _quarticDesc, _quarticId, _ranaDesc, _ranaId, _rastriginDesc, _rastriginId, _rosenbrokDesc, _rosenbrokId, _schwefelDesc, _schwefelId, _sineEnvelopeSineWaveDesc, _sineEnvelopeSineWaveId, _stepDesc, _stepId, _stretchedVSineWaveDesc, and _stretchedVSineWaveId.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00079         {
00080             switch (f)
00081             {
00082                 case _schwefelId:
00083                     return _schwefelDesc;
00084                 case _dejongId:
00085                     return _dejongDesc;
00086                 case _rosenbrokId:
00087                     return _rosenbrokDesc;
00088                 case _rastriginId:
00089                     return _rastriginDesc;
00090                 case _griewangkId:
00091                     return _griewangkDesc;
00092                 case _sineEnvelopeSineWaveId:
00093                     return _sineEnvelopeSineWaveDesc;
00094                 case _stretchedVSineWaveId:
00095                     return _stretchedVSineWaveDesc;
00096                 case _ackleysOneId:
00097                     return _ackleysOneDesc;
00098                 case _ackleysTwoId:
00099                     return _ackleysTwoDesc;
00100                case _eggHolderId:
00101                    return _eggHolderDesc;
00102                case _ranaId:
00103                    return _ranaDesc;
00104                case _pathologicalId:
00105                    return _pathologicalDesc;
00106                case _michalewiczId:
00107                    return _michalewiczDesc;
00108                case _mastersCosineWaveId:
00109                    return _mastersCosineWaveDesc;
00110                case _quarticId:
00111                    return _quarticDesc;
00112                case _levyId:
00113                    return _levyDesc;
00114                case _stepId:
00115                    return _stepDesc;
00116                case _alpineId:
00117                    return _alpineDesc;
00118                default:
00119                    return NULL;
00120            }
00121        }
```

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

- include/mfunctions.h

5.6 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)
- static size_t getCallCounter (unsigned int f)

  Returns the number of times the specified function id has been executed.
- static void resetCallCounters ()

  Resets all function call counters to zero.

5.6.1 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

T	Data type for function calculations

Definition at line 132 of file mfunctions.h.

5.6.2 Member Function Documentation

5.6.2.1 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

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

Returns

The result of the mathematical function

Definition at line 385 of file mfunctions.h.

References _ackleysOneId.

```
00386 {
00387     fCounterInc(_ackleysOneId);
00388
00389     T f = 0.0;
00390
00391     for (size_t i = 0; i < n - 1; i++)
00392     {
00393         T a = (static_cast<T>(1.0) / std::pow(static_cast<T>(M_E), static_cast<T>(0.2))) * std::sqrt(v[i]*v
      [i] + v[i+1]*v[i+1]);
00394         T b = static_cast<T>(3.0) * (std::cos(static_cast<T>(2.0) * v[i]) + std::sin(static_cast<T>(2.0) *
      v[i+1]));
00395         f += a + b;
00396     }
00397
00398     return f;
00399 }
```

5.6.2.2 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

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

Returns

The result of the mathematical function

Definition at line 411 of file mfunctions.h.

References _ackleysTwoId.

```
00412 {
00413     fCounterInc(_ackleysTwoId);
00414
00415     T f = 0.0;
00416
00417     for (size_t i = 0; i < n - 1; i++)
00418     {
00419         T a = static_cast<T>(20.0) / std::pow(static_cast<T>(M_E), static_cast<T>(0.2) * std::sqrt((v[i]*v[
      i] + v[i+1]*v[i+1]) / static_cast<T>(2.0)));
00420         T b = std::pow(static_cast<T>(M_E), static_cast<T>(0.5) *
00421             (std::cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + std::cos(static_cast<T>(2.0) *
      static_cast<T>(M_PI) * v[i+1])));
00422         f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
00423     }
00424
00425     return f;
00426 }
```

5.6.2.3 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 659 of file mfunctions.h.

References _alpineId.

```
00660 {
00661     fCounterInc(_alpineId);
00662
00663     T f = 0.0;
00664
00665     for (size_t i = 0; i < n; i++)
00666     {
00667         f += std::abs(v[i] * std::sin(v[i]) + static_cast<T>(0.1)*v[i]);
00668     }
00669
00670     return f;
00671 }
```

5.6.2.4 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 225 of file mfunctions.h.

References _dejongId.

```
00226 {
00227     fCounterInc(_dejongId);
00228
00229     T f = 0.0;
00230
00231     for (size_t i = 0; i < n; i++)
00232     {
00233         f += v[i] * v[i];
00234     }
00235
00236     return f;
00237 }
```

5.6.2.5 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

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

Returns

The result of the mathematical function

Definition at line 438 of file mfunctions.h.

References _eggHolderId.

```
00439 {
00440     fCounterInc(_eggHolderId);
00441
00442     T f = 0.0;
00443
00444     for (size_t i = 0; i < n - 1; i++)
00445     {
00446         T a = static_cast<T>(-1.0) * v[i] * std::sin(std::sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0
      ))));
00447         T b = (v[i+1] + static_cast<T>(47)) * std::sin(std::sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (
      v[i]/static_cast<T>(2.0)))));
00448         f += a - b;
00449     }
00450
00451     return f;
00452 }
```

5.6.2.6 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

f	Function id to execute
v	Vector as a T value array
n	Size of the vector 'v'
outResult	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 743 of file mfunctions.h.

```
00744 {
00745     auto fPtr = get(f);
00746     if (fPtr == nullptr) return false;
00747
00748     outResult = fPtr(v, n);
00749     return true;
00750 }
```

5.6.2.7 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

T	Data type to be used in the function's calculations

Parameters

f	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 685 of file mfunctions.h.

References _ackleysOneId, _ackleysTwoId, _alpineId, _dejongId, _eggHolderId, _griewangkId, _levyId, _masters↩CosineWaveId, _michalewiczId, _pathologicalId, _quarticId, _ranaId, _rastriginId, _rosenbrokId, _schwefelId, _↩sineEnvelopeSineWaveId, _stepId, and _stretchedVSineWaveId.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00686 {
00687     switch (f)
00688     {
00689         case _schwefelId:
00690             return Functions<T>::schwefel;
00691         case _dejongId:
00692             return Functions<T>::dejong;
00693         case _rosenbrokId:
00694             return Functions<T>::rosenbrok;
00695         case _rastriginId:
00696             return Functions<T>::rastrigin;
00697         case _griewangkId:
00698             return Functions<T>::griewangk;
00699         case _sineEnvelopeSineWaveId:
00700             return Functions<T>::sineEnvelopeSineWave;
00701         case _stretchedVSineWaveId:
00702             return Functions<T>::stretchedVSineWave;
00703         case _ackleysOneId:
00704             return Functions<T>::ackleysOne;
00705         case _ackleysTwoId:
00706             return Functions<T>::ackleysTwo;
00707         case _eggHolderId:
```

```
00708             return Functions<T>::eggHolder;
00709         case _ranaId:
00710             return Functions<T>::rana;
00711         case _pathologicalId:
00712             return Functions<T>::pathological;
00713         case _michalewiczId:
00714             return Functions<T>::michalewicz;
00715         case _mastersCosineWaveId:
00716             return Functions<T>::mastersCosineWave;
00717         case _quarticId:
00718             return Functions<T>::quartic;
00719         case _levyId:
00720             return Functions<T>::levy;
00721         case _stepId:
00722             return Functions<T>::step;
00723         case _alpineId:
00724             return Functions<T>::alpine;
00725         default:
00726             return nullptr;
00727     }
00728 }
```

5.6.2.8 getCallCounter()

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

Returns the number of times the specified function id has been executed.

Returns

size_t Number of times the given function id has been executed

Definition at line 758 of file mfunctions.h.

References _NUM_FUNCTIONS.

```
00759 {
00760     if (f == 0 || f > _NUM_FUNCTIONS)
00761         return 0;
00762
00763     return fCallCounters[f - 1];
00764 }
```

5.6.2.9 griewangk()

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

Function 5. Implementation of the Griewangk 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 300 of file mfunctions.h.

References _griewangkId.

```
00301 {
00302     fCounterInc(_griewangkId);
00303
00304     T sum = 0.0;
00305     T product = 0.0;
00306
00307     for (size_t i = 0; i < n; i++)
00308     {
00309         sum += (v[i] * v[i]) / static_cast<T>(4000.0);
00310     }
00311
00312     for (size_t i = 0; i < n; i++)
00313     {
00314         product *= std::cos(v[i] / std::sqrt(static_cast<T>(i + 1.0)));
00315     }
00316
00317     return static_cast<T>(1.0) + sum - product;
00318 }
```

5.6.2.10 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 601 of file mfunctions.h.

References _levyId.

```
00602 {
00603     fCounterInc(_levyId);
00604 
00605     T f = 0.0;
00606 
00607     for (size_t i = 0; i < n - 1; i++)
00608     {
00609         T a = w(v[i]) - static_cast<T>(1.0);
00610         a *= a;
00611         T b = std::sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
00612         b *= b;
00613         T c = w(v[n - 1]) - static_cast<T>(1.0);
00614         c *= c;
00615         T d = std::sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
00616         d *= d;
00617         f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
00618     }
00619 
00620     T e = std::sin(static_cast<T>(M_PI) * w(v[0]));
00621     return e*e + f;
00622 }
```

5.6.2.11 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 542 of file mfunctions.h.

References _mastersCosineWaveId.

```
00543 {
00544     fCounterInc(_mastersCosineWaveId);
00545 
00546     T f = 0.0;
00547 
00548     for (size_t i = 0; i < n - 1; i++)
00549     {
00550         T a = std::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]));
00551         T b = std::cos(static_cast<T>(4) * std::sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v
      [i+1]));
00552         f += a * b;
00553     }
00554 
00555     return static_cast<T>(-1.0) * f;
00556 }
```

5.6.2.12 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 518 of file mfunctions.h.

References _michalewiczId.

```
00519 {
00520     fCounterInc(_michalewiczId);
00521
00522     T f = 0.0;
00523
00524     for (size_t i = 0; i < n; i++)
00525     {
00526         f += std::sin(v[i]) * std::pow(std::sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)),
      static_cast<T>(20));
00527     }
00528
00529     return -1.0 * f;
00530 }
```

5.6.2.13 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 186 of file mfunctions.h.

```
00187 {
00188     return std::pow(x, static_cast<T>(1.0) / n);
00189 }
```

5.6.2.14 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 490 of file mfunctions.h.

References _pathologicalId.

```
00491 {
00492     fCounterInc(_pathologicalId);
00493
00494     T f = 0.0;
00495
00496     for (size_t i = 0; i < n - 1; i++)
00497     {
00498         T a = std::sin(std::sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
00499         a = (a*a) - static_cast<T>(0.5);
00500         T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
00501         b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
00502         f += static_cast<T>(0.5) + (a/b);
00503     }
00504
00505     return f;
00506 }
```

5.6.2.15 quartic()

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

Function 15. Implementation of the Quartic 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 568 of file mfunctions.h.

References _quarticId.

```
00569 {
00570     fCounterInc(_quarticId);
00571
00572     T f = 0.0;
00573
00574     for (size_t i = 0; i < n; i++)
00575     {
00576         f += (i+1) * v[i] * v[i] * v[i] * v[i];
00577     }
00578
00579     return f;
00580 }
```

5.6.2.16 rana()

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

Function 11. Implementation of the Rana 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 464 of file mfunctions.h.

References _ranaId.

```
00465 {
00466     fCounterInc(_ranaId);
00467
00468     T f = 0.0;
00469
```

```
00470     for (size_t i = 0; i < n - 1; i++)
00471     {
00472         T a = v[i] * std::sin(std::sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * std::cos(
      std::sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00473         T b = (v[i+1] + static_cast<T>(1.0)) * std::cos(std::sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1
      .0)))) * std::sin(std::sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00474         f += a + b;
00475     }
00476
00477     return f;
00478 }
```

5.6.2.17 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 276 of file mfunctions.h.

References _rastriginId.

```
00277 {
00278     fCounterInc(_rastriginId);
00279
00280     T f = 0.0;
00281
00282     for (size_t i = 0; i < n; i++)
00283     {
00284         f += (v[i] * v[i]) - (static_cast<T>(10.0) * std::cos(static_cast<T>(2.0) * static_cast<T>(M_PI) *
      v[i]));
00285     }
00286
00287     return static_cast<T>(10.0) * static_cast<T>(n) * f;
00288 }
```

5.6.2.18 resetCallCounters()

```
template<class T >
void mfunc::Functions< T >::resetCallCounters ( )  [static]
```

Resets all function call counters to zero.

Definition at line 770 of file mfunctions.h.

References _NUM_FUNCTIONS.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
00771 {
00772     for (size_t i = 0; i < _NUM_FUNCTIONS; i++)
00773         fCallCounters[i] = 0;
00774 }
```

5.6.2.19 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 249 of file mfunctions.h.

References _rosenbrokId.

```
00250 {
00251     fCounterInc(_rosenbrokId);
00252
00253     T f = 0.0;
00254
00255     for (size_t i = 0; i < n - 1; i++)
00256     {
00257         T a = ((v[i] * v[i]) - v[i+1]);
00258         T b = (static_cast<T>(1.0) - v[i]);
00259         f += static_cast<T>(100.0) * a * a;
00260         f += b * b;
00261     }
00262
00263     return f;
00264 }
```

5.6.2.20 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 201 of file mfunctions.h.

References _schwefelId.

```
00202 {
00203     fCounterInc(_schwefelId);
00204
00205     T f = 0.0;
00206
00207     for (size_t i = 0; i < n; i++)
00208     {
00209         f += (static_cast<T>(-1.0) * v[i]) * std::sin(std::sqrt(std::abs(v[i])));
00210     }
00211
00212     return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
00213 }
```

5.6.2.21 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

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

Returns

The result of the mathematical function

Definition at line 330 of file mfunctions.h.

References _sineEnvelopeSineWaveId.

```
00331 {
00332     fCounterInc(_sineEnvelopeSineWaveId);
00333
00334     T f = 0.0;
00335
```

```
00336     for (size_t i = 0; i < n - 1; i++)
00337     {
00338         T a = std::sin(v[i]*v[i] + v[i+1]*v[i+1] - static_cast<T>(0.5));
00339         a *= a;
00340         T b = (static_cast<T>(1.0) + static_cast<T>(0.001)*(v[i]*v[i] + v[i+1]*v[i+1]));
00341         b *= b;
00342         f += static_cast<T>(0.5) + (a / b);
00343     }
00344
00345     return static_cast<T>(-1.0) * f;
00346 }
```

5.6.2.22 step()

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

Function 17. Implementation of the Step 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 634 of file mfunctions.h.

References _stepId.

```
00635 {
00636     fCounterInc(_stepId);
00637
00638     T f = 0.0;
00639
00640     for (size_t i = 0; i < n; i++)
00641     {
00642         T a = std::abs(v[i]) + static_cast<T>(0.5);
00643         f += a * a;
00644     }
00645
00646     return f;
00647 }
```

5.6.2.23 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 358 of file mfunctions.h.

References _stretchedVSineWaveId.

```
00359 {
00360     fCounterInc(_stretchedVSineWaveId);
00361
00362     T f = 0.0;
00363
00364     for (size_t i = 0; i < n - 1; i++)
00365     {
00366         T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
00367         T b = std::sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0
    )));
00368         b *= b;
00369         f += a * b + static_cast<T>(1.0);
00370     }
00371
00372     return f;
00373 }
```

5.6.2.24 w()

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

Helper math function used in levy()

Definition at line 588 of file mfunctions.h.

```
00589 {
00590     return static_cast<T>(1.0) + (x - static_cast<T>(1.0)) / static_cast<T>(4.0);
00591 }
```

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

- include/mfunctions.h

5.7 mfunc::HarmonySearch< T > Class Template Reference

The HarmonySearch class runs the harmony search algorithm based on the parameters passed to the run() method.

`#include <harmsearch.h>`

Public Member Functions

- HarmonySearch ()

  Construct a new HarmonySearch object.
- ~HarmonySearch ()=default
- int run (HSParams< T > p)

  Runs the harmony search algorithm with the given parameters.

5.7.1 Detailed Description

template<class T>
class mfunc::HarmonySearch< T >

The HarmonySearch class runs the harmony search algorithm based on the parameters passed to the run() method.

Template Parameters

T	Data type used by the algorithm

Definition at line 77 of file harmsearch.h.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 HarmonySearch()

```
template<class T >
mfunc::HarmonySearch< T >::HarmonySearch ( )
```

Construct a new HarmonySearch object.

Template Parameters

T	Data type used by the algorithm

Definition at line 99 of file harmsearch.h.

```
00100     : seed(), engine(seed()), rchance(0, 1), rrange(-1, 1)
00101 {
00102 }
```

5.7.2.2 ~HarmonySearch()

```
template<class T>
mfunc::HarmonySearch< T >::~HarmonySearch ( )  [default]
```

5.7.3 Member Function Documentation

5.7.3.1 run()

```
template<class T >
int mfunc::HarmonySearch< T >::run (
            HSParams< T > p )
```

Runs the harmony search algorithm with the given parameters.

Template Parameters

T	Data type used by the algorithm

Parameters

p	Parameters for the search algorithm

Returns

Returns a non-zero error code on failure, or zero on success

Definition at line 112 of file harmsearch.h.

References mfunc::HSParams< T >::bestFitnessTable, mfunc::HSParams< T >::bw, mfunc::HSParams< T >::fitTableCol, mfunc::HSParams< T >::fMaxBound, mfunc::HSParams< T >::fMinBound, mfunc::HSParams< T >::fPtr, mfunc::HSParams< T >::hmcr, mfunc::HSParams< T >::iterations, mfunc::HSParams< T >::mainPop, mfunc::HSParams< T >::par, mfunc::HSParams< T >::popFile, POPFILE_GEN_PATTERN, and mfunc::HSParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testHS().

```
00113 {
00114     if (p.mainPop == nullptr || p.fPtr == nullptr)
00115         return 1;
00116
00117     // Get population information
00118     const size_t popSize = p.mainPop->getPopulationSize();
00119     const size_t dimSize = p.mainPop->getDimensionsSize();
00120
00121     T* solBuffer = util::allocArray<T>(dimSize);
00122     if (solBuffer == nullptr)
00123         return 2;
00124
00125     // Generate random population vectors
00126     if (!p.mainPop->generate(p.fMinBound, p.fMaxBound))
00127         return 3;
00128
00129     // Calculate fitness values for entire population
00130     if (!p.mainPop->calcAllFitness(p.fPtr))
00131         return 4;
00132
00133     // Sort fitness from best to worst
00134     p.mainPop->sortFitnessAscend();
00135
00136     for (unsigned int iter = 0; iter < p.iterations; iter++)
00137     {
00138         // Generate new solution
00139         adjustPitch(p, solBuffer, dimSize);
```

```
00140
00141         // Calculate the new fitness, and replace worst if new solution is better
00142         T newAesthetic = p.fPtr(solBuffer, dimSize);
00143         T oldAesthetic = p.mainPop->getFitness(popSize - 1);
00144         if (newAesthetic < oldAesthetic)
00145         {
00146             p.mainPop->copyPopulation(solBuffer, popSize - 1);
00147             p.mainPop->setFitness(popSize - 1, newAesthetic);
00148         }
00149
00150         // Resort population
00151         p.mainPop->sortFitnessAscend();
00152
00153         // Store best fitness value for this iteration
00154         if (p.bestFitnessTable != nullptr)
00155             p.bestFitnessTable->setEntry(iter, p.fitTableCol, p.mainPop->getFitness(0));
00156
00157         // Store worst fitness value for this iteration
00158         if (p.worstFitnessTable != nullptr)
00159             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.mainPop->getFitness(popSize - 1));
00160
00161         // Dump population vectors to a file
00162         if (!p.popFile.empty())
00163             p.mainPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
      POPFILE_GEN_PATTERN), std::to_string(iter)));
00164     }
00165
00166     util::releaseArray<T>(solBuffer);
00167
00168     return 0;
00169 }
```

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

- include/harmsearch.h

5.8 mfunc::HSParams< T > Struct Template Reference

The HSParams struct contains various parameters that are required to be passed to the HarmonySearch.run() method.

`#include <harmsearch.h>`

Public Member Functions

- HSParams ()

  Construct a new HSParams object.

Public Attributes

- std::string popFile
- mdata::DataTable< T > * bestFitnessTable
- mdata::DataTable< T > * worstFitnessTable
- size_t fitTableCol
- mdata::Population< T > * mainPop
- mfuncPtr< T > fPtr
- T fMinBound
- T fMaxBound
- unsigned int iterations
- double hmcr
- double par
- double bw

5.8.1 Detailed Description

template<class T>
struct mfunc::HSParams< T >

The HSParams struct contains various parameters that are required to be passed to the HarmonySearch.run() method.

Template Parameters

T	Data type used by the search algorithm

Definition at line 35 of file harmsearch.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 HSParams()

```
template<class T>
mfunc::HSParams< T >::HSParams ( )  [inline]
```

Construct a new HSParams object.

Definition at line 53 of file harmsearch.h.

```
00054         {
00055             popFile = "";
00056             bestFitnessTable = nullptr;
00057             worstFitnessTable = nullptr;
00058             fitTableCol = 0;
00059             mainPop = nullptr;
00060             fPtr = nullptr;
00061             fMinBound = 0;
00062             fMaxBound = 0;
00063             iterations = 0;
00064             hmcr = 0;
00065             par = 0;
00066             bw = 0;
00067         }
```

5.8.3 Member Data Documentation

5.8.3.1 bestFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::HSParams< T >::bestFitnessTable
```

Definition at line 38 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.2 bw

```
template<class T>
double mfunc::HSParams< T >::bw
```

Definition at line 48 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.3 fitTableCol

```
template<class T>
size_t mfunc::HSParams< T >::fitTableCol
```

Definition at line 40 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.4 fMaxBound

```
template<class T>
T mfunc::HSParams< T >::fMaxBound
```

Definition at line 44 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.5 fMinBound

```
template<class T>
T mfunc::HSParams< T >::fMinBound
```

Definition at line 43 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.6 fPtr

```
template<class T>
mfuncPtr<T> mfunc::HSParams< T >::fPtr
```

Definition at line 42 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.7 hmcr

```
template<class T>
double mfunc::HSParams< T >::hmcr
```

Definition at line 46 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.8 iterations

```
template<class T>
unsigned int mfunc::HSParams< T >::iterations
```

Definition at line 45 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.9 mainPop

```
template<class T>
mdata::Population<T>* mfunc::HSParams< T >::mainPop
```

Definition at line 41 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.10 par

```
template<class T>
double mfunc::HSParams< T >::par
```

Definition at line 47 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.11 popFile

```
template<class T>
std::string mfunc::HSParams< T >::popFile
```

Definition at line 37 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

5.8.3.12 worstFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::HSParams< T >::worstFitnessTable
```

Definition at line 39 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run(), and mfunc::Experiment< T >::testHS().

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

- include/harmsearch.h

5.9 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, std::string defVal="")
  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, T defVal={})

5.9.1 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.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 IniReader()

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

Construct a new IniReader object.

Definition at line 21 of file inireader.cpp.

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

5.9.2.2 ~IniReader()

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

Destroys the IniReader object.

Definition at line 28 of file inireader.cpp.

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

5.9.3 Member Function Documentation

5.9.3.1 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

section	std::string containing the section name
entry	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 }
```

5.9.3.2 getEntry()

```
std::string IniReader::getEntry (
            std::string section,
            std::string entry,
            std::string defVal = "" )
```

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

Parameters

section	std::string containing the section name
entry	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 >::testHS().

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

5.9.3.3 getEntryAs()

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

Definition at line 57 of file inireader.h.

References getEntry().

Referenced by mfunc::Experiment< T >::init(), and mfunc::Experiment< T >::testHS().

```
00057                                                                   {})
00058         {
00059             std::stringstream ss(getEntry(section, entry, std::to_string(defVal)));
00060             T retVal;
00061             ss >> retVal;
00062             return retVal;
00063         }
```

5.9.3.4 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

filePath	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 }
```

5.9.3.5 sectionExists()

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

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

Parameters

section	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

5.10 mfunc::Particle< T > Struct Template Reference

The Particle struct is a simple data structure used to store the global best particle along with it's fitness.

`#include <partswarm.h>`

Public Member Functions

- Particle ()

Public Attributes

- T * vector
- T fitness

5.10.1 Detailed Description

template<class T>
struct mfunc::Particle< T >

The Particle struct is a simple data structure used to store the global best particle along with it's fitness.

Template Parameters

T	

Definition at line 34 of file partswarm.h.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 Particle()

```
template<class T>
mfunc::Particle< T >::Particle ( )  [inline]
```

Definition at line 39 of file partswarm.h.

```
00040             : vector(nullptr), fitness(0)
00041         {
00042         }
```

5.10.3 Member Data Documentation

5.10.3.1 fitness

```
template<class T>
T mfunc::Particle< T >::fitness
```

Definition at line 37 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run().

5.10.3.2 vector

```
template<class T>
T* mfunc::Particle< T >::vector
```

Definition at line 36 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run().

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

- include/partswarm.h

5.11 mfunc::ParticleSwarm< T > Class Template Reference

The ParticleSwarm class runs the particle swarm algorithm with the given parameters passed to the run() method.

`#include <partswarm.h>`

Public Member Functions

- ParticleSwarm ()

  Construct a new ParticleSwarm object.
- ~ParticleSwarm ()=default
- int run (PSParams< T > params)

  Runs the particle swarm algorithm with the given parameters.

5.11.1 Detailed Description

template<class T>
class mfunc::ParticleSwarm< T >

The ParticleSwarm class runs the particle swarm algorithm with the given parameters passed to the run() method.

Template Parameters

T	Data type used by the search algorithm

Definition at line 96 of file partswarm.h.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 ParticleSwarm()

```
template<class T >
mfunc::ParticleSwarm< T >::ParticleSwarm ( )
```

Construct a new ParticleSwarm object.

Template Parameters

T	Data type used by the search algorithm

Definition at line 117 of file partswarm.h.

```
00118     : seed(), engine(seed()), rchance(0, 1)
00119 {
00120 }
```

5.11.2.2 ~ParticleSwarm()

```
template<class T>
mfunc::ParticleSwarm< T >::~ParticleSwarm ( )  [default]
```

5.11.3 Member Function Documentation

5.11.3.1 run()

```
template<class T >
int mfunc::ParticleSwarm< T >::run (
            PSParams< T > p )
```

Runs the particle swarm algorithm with the given parameters.

Template Parameters

T	Data type used by the search algorithm

Parameters

p	Parameters used by the search algorithm

Returns

Returns a non-zero error code on failure, or zero on success

Definition at line 130 of file partswarm.h.

References mfunc::PSParams< T >::bestFitnessTable, mfunc::PSParams< T >::c1, mfunc::PSParams< T >::c2, mfunc::Particle< T >::fitness, mfunc::PSParams< T >::fitTableCol, mfunc::PSParams< T >::fMaxBound, mfunc::PSParams< T >::fMinBound, mfunc::PSParams< T >::fPtr, mfunc::PSParams< T >::iterations, mfunc::PSParams< T >::k, mfunc::PSParams< T >::mainPop, mfunc::PSParams< T >::pbPop, mfunc::PSParams< T >::popFile, POPFILE_GEN_PATTERN, mfunc::Particle< T >::vector, and mfunc::PSParams< T >::worstFitnessTable.

Referenced by mfunc::Experiment< T >::testPS().

```
00131 {
00132     if (p.mainPop == nullptr || p.pbPop == nullptr || p.fPtr == nullptr)
00133         return 1;
00134
00135     // Get population information
00136     const size_t popSize = p.mainPop->getPopulationSize();
00137     const size_t dimSize = p.mainPop->getDimensionsSize();
00138
00139     if (popSize != p.pbPop->getPopulationSize() ||
```

```
00140         dimSize != p.pbPop->getDimensionsSize())
00141         return 2;
00142
00143
00144     // Construct global best particle and allocate gBest vector
00145     Particle<T> globalBest;
00146     globalBest.vector = util::allocArray<T>(dimSize);
00147
00148     // Allocate velocity matrix
00149     T** velocityMatrix = util::allocMatrix<T>(popSize, dimSize);
00150
00151     if (globalBest.vector == nullptr || velocityMatrix == nullptr)
00152         return 3;
00153
00154     if (!p.mainPop->generate(p.fMinBound, p.fMaxBound))
00155         return 4;
00156
00157     if (!p.mainPop->calcAllFitness(p.fPtr))
00158         return 5;
00159
00160     if (!p.pbPop->copyAllFrom(p.mainPop))
00161         return 6;
00162
00163
00164     // Randomize the velocities for all particles
00165     randomizeVelocity(velocityMatrix, popSize, dimSize, p.fMinBound, p.fMaxBound);
00166
00167     auto bestFitIndex = p.mainPop->getBestFitnessIndex();
00168     util::copyArray<T>(p.mainPop->getPopulationPtr(bestFitIndex), globalBest.vector, dimSize);
00169     globalBest.fitness = p.mainPop->getFitness(bestFitIndex);
00170
00171     for (unsigned int iter = 0; iter < p.iterations; iter++)
00172     {
00173         for (size_t pIndex = 0; pIndex < popSize; pIndex++)
00174         {
00175             // Update the particles and their velocities
00176             updateParticle(p, globalBest, velocityMatrix, pIndex);
00177         }
00178
00179         // Get the index of current the best solution, and the associated fitness
00180         bestFitIndex = p.mainPop->getBestFitnessIndex();
00181         T bestFitVal = p.mainPop->getFitness(bestFitIndex);
00182
00183         // Update global best if current best is better
00184         if (bestFitVal < globalBest.fitness)
00185         {
00186             util::copyArray<T>(p.mainPop->getPopulationPtr(bestFitIndex), globalBest.vector, dimSize);
00187             globalBest.fitness = bestFitVal;
00188         }
00189
00190         // Store best fitness for this iteration
00191         if (p.bestFitnessTable != nullptr)
00192             p.bestFitnessTable->setEntry(iter, p.fitTableCol, globalBest.fitness);
00193
00194         // Store worst fitness for this iteration
00195         if (p.worstFitnessTable != nullptr)
00196             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.mainPop->getWorstFitness());
00197
00198         // Dump population vectors to file
00199         if (!p.popFile.empty())
00200             p.mainPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
      POPFILE_GEN_PATTERN), std::to_string(iter)));
00201     }
00202
00203     util::releaseArray<T>(globalBest.vector);
00204     util::releaseMatrix<T>(velocityMatrix, popSize);
00205
00206     return 0;
00207 }
```

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

- include/partswarm.h

5.12 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.
- T * getBestPopulationPtr ()
- 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 generateSingle (size_t popIndex, T minBound, T maxBound)
- 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.
- bool calcAllFitness (mfunc::mfuncPtr< T > funcPtr)
- 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.
- T * getBestFitnessPtr ()

  Returns a pointer to the current best fitness value.
- size_t getBestFitnessIndex ()

  Returns the index of the current best fitness value.
- T getBestFitness ()
- size_t getWorstFitnessIndex ()
- T getWorstFitness ()
- void sortFitnessAscend ()
- void sortFitnessDescend ()
- bool copyFrom (Population< T > *srcPtr, size_t srcIndex, size_t destIndex)
- bool copyAllFrom (Population< T > *srcPtr)
- bool copyPopulation (T *src, size_t destIndex)
- 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.
- bool outputPopulationCsv (std::string filePath)

5.12.1 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

T	Data type of the population.

Definition at line 29 of file population.h.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 Population()

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

Construct a new Population object.

Template Parameters

T	Data type of the population.

Parameters

pSize	Size of the population.
dimensions	Dimensions of the population.

Definition at line 28 of file population.cpp.

```
00029     : popMatrix(nullptr), popSize(pSize), popDim(dimensions), rdev(), rgen(rdev())
00030 {
00031     if (!allocPopMatrix() || !allocPopFitness())
00032         throw std::bad_alloc();
00033 }
```

5.12.2.2 ~Population()

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

Destroy Population object.

Template Parameters

T	Data type of the population.

Definition at line 41 of file population.cpp.

```
00042 {
00043     releasePopMatrix();
00044     releasePopFitness();
00045 }
```

5.12.3 Member Function Documentation

5.12.3.1 calcAllFitness()

```
template<class T >
bool Population::calcAllFitness (
            mfunc::mfuncPtr< T > funcPtr )
```

Definition at line 197 of file population.cpp.

References mdata::Population< T >::calcFitness().

```
00198 {
00199     for (size_t i = 0; i < popSize; i++)
00200     {
00201         if (!calcFitness(i, funcPtr))
00202             return false;
00203     }
00204
00205     return true;
00206 }
```

5.12.3.2 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

T	Data type of the population.

Parameters

popIndex	Index of the population vector you wish to set the fitness for.
funcPtr	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 187 of file population.cpp.

Referenced by mdata::Population< T >::calcAllFitness().

```
00188 {
00189     if (popFitness == nullptr || popIndex >= popSize) return false;
00190
00191     popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);
00192
00193     return true;
00194 }
```

5.12.3.3 copyAllFrom()

```
template<class T >
bool Population::copyAllFrom (
            Population< T > * srcPtr )
```

Definition at line 328 of file population.cpp.

References mdata::Population< T >::copyFrom(), mdata::Population< T >::getDimensionsSize(), and mdata::Population< T >::getPopulationSize().

```
00329 {
00330     if (srcPtr == nullptr) return false;
00331
00332     const size_t srcSize = srcPtr->getPopulationSize();
00333     const size_t srcDim = srcPtr->getDimensionsSize();
00334
00335     if (srcSize != popSize || srcDim != popDim)
00336         return false;
00337
00338     for (size_t i = 0; i < popSize; i++)
00339     {
00340         if (!copyFrom(srcPtr, i, i))
00341             return false;
00342     }
00343
00344     return true;
00345 }
```

5.12.3.4 copyFrom()

```
template<class T >
bool Population::copyFrom (
            Population< T > * srcPtr,
            size_t srcIndex,
            size_t destIndex )
```

Definition at line 309 of file population.cpp.

References mdata::Population< T >::getDimensionsSize(), mdata::Population< T >::getFitness(), mdata::Population< T >::getPopulationPtr(), and mdata::Population< T >::setFitness().

Referenced by mdata::Population< T >::copyAllFrom().

```
00310 {
00311     if (srcPtr == nullptr) return false;
00312
00313     const size_t srcDim = srcPtr->getDimensionsSize();
00314     if (srcDim != popDim) return false;
00315
00316     T* srcVector = srcPtr->getPopulationPtr(srcIndex);
00317     T* destVector = getPopulationPtr(destIndex);
00318
00319     if (srcVector == nullptr || destVector == nullptr) return false;
00320
00321     copyArray<T>(srcVector, destVector, popDim);
00322     setFitness(destIndex, srcPtr->getFitness(srcIndex));
00323
00324     return true;
00325 }
```

5.12.3.5 copyPopulation()

```
template<class T >
bool Population::copyPopulation (
            T * src,
            size_t destIndex )
```

Definition at line 348 of file population.cpp.

References mdata::Population< T >::getPopulationPtr().

```
00349 {
00350     T* destVect = getPopulationPtr(destIndex);
00351     if (destVect == nullptr)
00352         return false;
00353
00354     for (size_t i = 0; i < popDim; i++)
00355     {
00356         destVect[i] = src[i];
00357     }
00358
00359     return true;
00360 }
```

5.12.3.6 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

T	Data type of the population.

Parameters

minBound	The minimum bound for a population value.
maxBound	The maximum bound for a population value.

Returns

Returns true of the population was succesfully generated, otherwise false.

Definition at line 116 of file population.cpp.

```
00117 {
00118     if (popMatrix == nullptr) return false;
00119
00120     // Set up a uniform distribution for the random number generator with the correct function bounds
00121     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00122
00123     // Generate values for all vectors in popMatrix
00124     for (size_t s = 0; s < popSize; s++)
00125     {
00126         for (size_t d = 0; d < popDim; d++)
00127         {
00128             T rand = (T)dist(rgen);
00129             popMatrix[s][d] = rand;
00130         }
00131     }
00132
00133     // Reset popFitness values to 0
00134     initArray<T>(popFitness, popSize, (T)0.0);
00135
00136     return true;
00137 }
```

5.12.3.7 generateSingle()

```
template<class T >
bool Population::generateSingle (
            size_t popIndex,
            T minBound,
            T maxBound )
```

Definition at line 140 of file population.cpp.

```
00141 {
00142     if (popMatrix == nullptr || popIndex >= popSize) return false;
00143
00144     // Set up a uniform distribution for the random number generator with the correct function bounds
00145     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00146
00147     for (size_t d = 0; d < popDim; d++)
00148     {
00149         T rand = (T)dist(rgen);
00150         popMatrix[popIndex][d] = rand;
00151     }
00152
00153     popFitness[popIndex] = 0;
00154
00155     return true;
00156 }
```

5.12.3.8 getBestFitness()

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

Definition at line 271 of file population.cpp.

References mdata::Population< T >::getBestFitnessIndex(), and mdata::Population< T >::getFitness().

```
00272 {
00273     return getFitness(getBestFitnessIndex());
00274 }
```

5.12.3.9 getBestFitnessIndex()

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

Returns the index of the current best fitness value.

Template Parameters

T	Data type of the population.

Returns

size_t Index of the best fitness value

Definition at line 257 of file population.cpp.

Referenced by mdata::Population< T >::getBestFitness(), mdata::Population< T >::getBestFitnessPtr(), and mdata::Population< T >::getBestPopulationPtr().

```
00258 {
00259     size_t bestIndex = 0;
00260
00261     for (size_t i = 1; i < popSize; i++)
00262     {
00263         if (popFitness[i] < popFitness[bestIndex])
00264             bestIndex = i;
00265     }
00266
00267     return bestIndex;
00268 }
```

5.12.3.10 getBestFitnessPtr()

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

Returns a pointer to the current best fitness value.

Template Parameters

T	Data type of the population.

Returns

T* Pointer to the best fitness value

Definition at line 245 of file population.cpp.

References mdata::Population< T >::getBestFitnessIndex().

```
00246 {
00247     return &popFitness[getBestFitnessIndex()];
00248 }
```

5.12.3.11 getBestPopulationPtr()

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

Definition at line 100 of file population.cpp.

References mdata::Population< T >::getBestFitnessIndex(), and mdata::Population< T >::getPopulationPtr().

```
00101 {
00102     return getPopulationPtr(getBestFitnessIndex());
00103 }
```

5.12.3.12 getDimensionsSize()

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

Returns the dimensions of the population.

Template Parameters

T	Data type of the population.

Returns

The number of dimensions in the population.

Definition at line 79 of file population.cpp.

Referenced by mdata::Population< T >::copyAllFrom(), and mdata::Population< T >::copyFrom().

```
00080 {
00081     return popDim;
00082 }
```

5.12.3.13 getFitness()

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

Returns the fitness value for a specific population vector index.

Template Parameters

T	Data type of the population.

Parameters

popIndex	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 216 of file population.cpp.

Referenced by mdata::Population< T >::copyFrom(), mdata::Population< T >::getBestFitness(), and mdata::Population< T >::getWorstFitness().

```
00217 {
00218     if (popFitness == nullptr || popIndex >= popSize) return 0;
00219 
00220     return popFitness[popIndex];
00221 }
```

5.12.3.14 getFitnessPtr()

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

Returns the fitness value for a specific population vector index.

Template Parameters

T	Data type of the population.

Parameters

popIndex	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 231 of file population.cpp.

```
00232 {
00233     if (popFitness == nullptr || popIndex >= popSize) return 0;
00234
00235     return &popFitness[popIndex];
00236 }
```

5.12.3.15 getPopulationPtr()

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

Returns an array for the population with the given index.

Template Parameters

T	Data type of the population.

Parameters

popIndex	Index of the population vector you wish to retrieve.

Returns

Pointer to population vector array at the given index.

Definition at line 92 of file population.cpp.

Referenced by mdata::Population< T >::copyFrom(), mdata::Population< T >::copyPopulation(), and mdata::Population< T >::getBestPopulationPtr().

```
00093 {
00094     if (popFitness == nullptr || popIndex >= popSize) return nullptr;
00095
00096     return popMatrix[popIndex];
00097 }
```

5.12.3.16 getPopulationSize()

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

Returns the size of the population.

Template Parameters

T	Data type of the population.

Returns

The size of the population.

Definition at line 67 of file population.cpp.

Referenced by mdata::Population< T >::copyAllFrom().

```
00068 {
00069     return popSize;
00070 }
```

5.12.3.17 getWorstFitness()

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

Definition at line 291 of file population.cpp.

References mdata::Population< T >::getFitness(), and mdata::Population< T >::getWorstFitnessIndex().

```
00292 {
00293     return getFitness(getWorstFitnessIndex());
00294 }
```

5.12.3.18 getWorstFitnessIndex()

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

Definition at line 277 of file population.cpp.

Referenced by mdata::Population< T >::getWorstFitness().

```
00278 {
00279     size_t worstIndex = 0;
00280
00281     for (size_t i = 1; i < popSize; i++)
00282     {
00283         if (popFitness[i] > popFitness[worstIndex])
00284             worstIndex = i;
00285     }
00286
00287     return worstIndex;
00288 }
```

5.12.3.19 isReady()

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

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

Template Parameters

T	Data type of the population.

Returns

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

Definition at line 55 of file population.cpp.

```
00056 {
00057     return popMatrix != nullptr && popFitness != nullptr;
00058 }
```

5.12.3.20 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

T	Data type of the population.

Parameters

outStream	Output stream to write the data to.
delim	Delimiter characters to separate columns.
lineBreak	Delimiter characters to separate rows.

Definition at line 413 of file population.cpp.

```
00414 {
00415     if (popFitness == nullptr) return;
00416
00417     for (size_t j = 0; j < popSize; j++)
00418     {
```

```
00419         outStream << popFitness[j];
00420             if (j < popSize - 1)
00421                 outStream << delim;
00422     }
00423
00424     if (lineBreak != nullptr)
00425         outStream << lineBreak;
00426 }
```

5.12.3.21 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

T	Data type of the population.

Parameters

outStream	Output stream to write the data to.
delim	Delimiter characters to separate columns.
lineBreak	Delimiter characters to separate rows.

Definition at line 371 of file population.cpp.

Referenced by mdata::Population< T >::outputPopulationCsv().

```
00372 {
00373     if (popMatrix == nullptr) return;
00374
00375     for (size_t j = 0; j < popSize; j++)
00376     {
00377         for (size_t k = 0; k < popDim; k++)
00378         {
00379             outStream << popMatrix[j][k];
00380             if (k < popDim - 1)
00381                 outStream << delim;
00382         }
00383
00384         outStream << lineBreak;
00385     }
00386 }
```

5.12.3.22 outputPopulationCsv()

```
template<class T >
bool Population::outputPopulationCsv (
             std::string filePath )
```

Definition at line 389 of file population.cpp.

References mdata::Population< T >::outputPopulation().

```
00390 {
00391     static const char* delim = ",";
00392     static const char* newline = "\n";
00393
00394     std::ofstream file;
00395     file.open(filePath, std::ios::out | std::ios::trunc);
00396     if (!file.good()) return false;
00397
00398     outputPopulation(file, delim, newline);
00399     file.close();
00400
00401     return true;
00402 }
```

5.12.3.23 setFitness()

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

Sets the fitness value for a specific population vector index.

Template Parameters

T	Data type of the population.

Parameters

popIndex	Index of the population vector you wish to set the fitness for.
value	The value of the fitness.

Returns

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

Definition at line 167 of file population.cpp.

Referenced by mdata::Population< T >::copyFrom().

```
00168 {
00169     if (popFitness == nullptr || popIndex >= popSize) return false;
00170
00171     popFitness[popIndex] = value;
00172
00173     return true;
00174 }
```

5.12.3.24 sortFitnessAscend()

```
template<class T >
void Population::sortFitnessAscend ( )
```

Definition at line 297 of file population.cpp.

```
00298 {
00299     qs_fit_ascend(0, popSize - 1);
00300 }
```

5.12.3.25 sortFitnessDescend()

```
template<class T >
void Population::sortFitnessDescend ( )
```

Definition at line 303 of file population.cpp.

```
00304 {
00305     qs_fit_descend(0, popSize - 1);
00306 }
```

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

- include/population.h
- src/population.cpp

5.13 mfunc::PSParams< T > Struct Template Reference

The PSParams struct contains various parameters that are required to be passed to the ParticleSwarm.run() method.

```
#include <partswarm.h>
```

Public Member Functions

- PSParams ()

  Construct a new PSParams object.

Public Attributes

- std::string popFile
- mdata::DataTable< T > * bestFitnessTable
- mdata::DataTable< T > * worstFitnessTable
- size_t fitTableCol
- mdata::Population< T > * mainPop
- mdata::Population< T > * pbPop
- mfuncPtr< T > fPtr
- T fMinBound
- T fMaxBound
- unsigned int iterations
- double c1
- double c2
- double k

5.13.1 Detailed Description

template<class T>
struct mfunc::PSParams< T >

The PSParams struct contains various parameters that are required to be passed to the ParticleSwarm.run() method.

Template Parameters

T	Data type used by the search algorithm

Definition at line 52 of file partswarm.h.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 PSParams()

```
template<class T>
mfunc::PSParams< T >::PSParams ( )  [inline]
```

Construct a new PSParams object.

Definition at line 71 of file partswarm.h.

```
00072         {
00073             popFile = "";
00074             bestFitnessTable = nullptr;
00075             worstFitnessTable = nullptr;
00076             fitTableCol = 0;
00077             mainPop = nullptr;
00078             pbPop = nullptr;
00079             fPtr = nullptr;
00080             fMinBound = 0;
00081             fMaxBound = 0;
00082             iterations = 0;
00083             c1 = 0;
00084             c2 = 0;
00085             k = 0;
00086         }
```

5.13.3 Member Data Documentation

5.13.3.1 bestFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::PSParams< T >::bestFitnessTable
```

Definition at line 55 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.2 c1

```
template<class T>
double mfunc::PSParams< T >::c1
```

Definition at line 64 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testHS().

5.13.3.3 c2

```
template<class T>
double mfunc::PSParams< T >::c2
```

Definition at line 65 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testHS().

5.13.3.4 fitTableCol

```
template<class T>
size_t mfunc::PSParams< T >::fitTableCol
```

Definition at line 57 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.5 fMaxBound

```
template<class T>
T mfunc::PSParams< T >::fMaxBound
```

Definition at line 62 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.6 fMinBound

```
template<class T>
T mfunc::PSParams< T >::fMinBound
```

Definition at line 61 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.7 fPtr

```
template<class T>
mfuncPtr<T> mfunc::PSParams< T >::fPtr
```

Definition at line 60 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.8 iterations

```
template<class T>
unsigned int mfunc::PSParams< T >::iterations
```

Definition at line 63 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.9 k

```
template<class T>
double mfunc::PSParams< T >::k
```

Definition at line 66 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testHS().

5.13.3.10 mainPop

```
template<class T>
mdata::Population<T>* mfunc::PSParams< T >::mainPop
```

Definition at line 58 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.11 pbPop

```
template<class T>
mdata::Population<T>* mfunc::PSParams< T >::pbPop
```

Definition at line 59 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.12 popFile

```
template<class T>
std::string mfunc::PSParams< T >::popFile
```

Definition at line 54 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

5.13.3.13 worstFitnessTable

```
template<class T>
mdata::DataTable<T>* mfunc::PSParams< T >::worstFitnessTable
```

Definition at line 56 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run(), and mfunc::Experiment< T >::testPS().

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

- include/partswarm.h

5.14 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

5.14.1 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.

5.14.2 Member Data Documentation

5.14.2.1 max

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

Definition at line 38 of file experiment.h.

Referenced by mfunc::Experiment< T >::testHS().

5.14.2.2 min

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

Definition at line 37 of file experiment.h.

Referenced by mfunc::Experiment< T >::testHS().

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

- include/experiment.h

5.15 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 ()

5.15.1 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.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 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 }
```

5.15.2.2 ~ThreadPool()

```
ThreadPool::~ThreadPool ( )  [inline]
```

Definition at line 117 of file threadpool.h.

References stopAndJoinAll().

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

5.15.3 Member Function Documentation

5.15.3.1 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 >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

```
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 }
```

5.15.3.2 stopAndJoinAll()

```
void ThreadPool::stopAndJoinAll ( )  [inline]
```

Definition at line 122 of file threadpool.h.

Referenced by mfunc::Experiment< T >::testHS(), 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 6

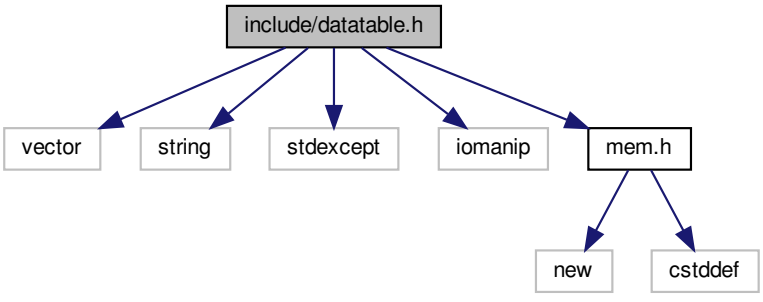
File Documentation

6.1 include/datatable.h File Reference

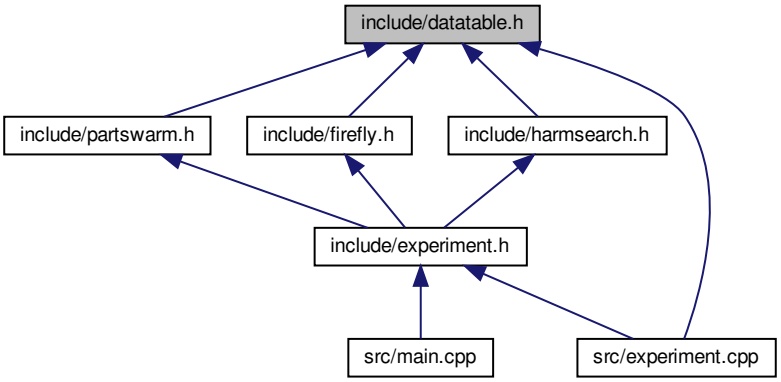
Header file for the DataTable class, which represents a spreadsheet/table of values that can easily be exported to a ∗.csv file.

```
#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

6.1.1 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.

6.2 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
00082         void clearData()
00083         {
00084             util::initMatrix<T>(dataMatrix, rows, cols, 0);
00085         }
00086
00093         std::string getColLabel(size_t colIndex)
00094         {
00095             if (colIndex >= colLabels.size())
00096                 throw std::out_of_range("Column index out of range");
00097
00098             return colLabels[colIndex];
00099         }
00100
00107         void setColLabel(size_t colIndex, std::string newLabel)
00108         {
00109              if (colIndex >= colLabels.size())
00110                 throw std::out_of_range("Column index out of range");
00111
00112             colLabels[colIndex] = newLabel;
00113         }
00114
00122         T getEntry(size_t row, size_t col)
00123         {
00124             if (dataMatrix == nullptr)
00125                 throw std::runtime_error("Data matrix not allocated");
00126             if (row >= rows)
00127                 throw std::out_of_range("Table row out of range");
00128             else if (col >= cols)
00129                 throw std::out_of_range("Table column out of range");
00130
00131             return dataMatrix[row][col];
00132         }
00133
00141         void setEntry(size_t row, size_t col, T val)
00142         {
00143             if (dataMatrix == nullptr)
00144                 throw std::runtime_error("Data matrix not allocated");
00145             if (row >= rows)
00146                 throw std::out_of_range("Table row out of range");
00147             else if (col >= cols)
00148                 throw std::out_of_range("Table column out of range");
00149
00150             dataMatrix[row][col] = val;
00151         }
00152
00160         bool exportCSV(const char* filePath)
00161         {
00162             if (dataMatrix == nullptr) return false;
00163
```

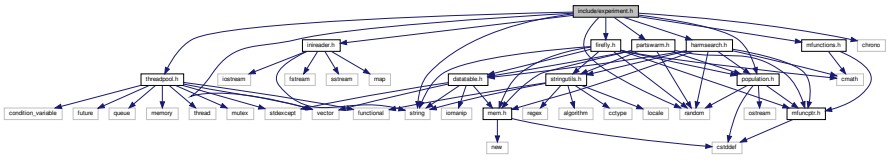
```
00164             using namespace std;
00165             ofstream outFile;
00166             outFile.open(filePath, ofstream::out | ofstream::trunc);
00167             if (!outFile.good()) return false;
00168 
00169             // Print column labels
00170             for (unsigned int c = 0; c < cols; c++)
00171             {
00172                 outFile << colLabels[c];
00173                 if (c < cols - 1) outFile << ",";
00174             }
00175 
00176             outFile << endl;
00177 
00178             // Print data rows
00179             for (unsigned int r = 0; r < rows; r++)
00180             {
00181                 for (unsigned int c = 0; c < cols; c++)
00182                 {
00183                     outFile << std::setprecision(8) << dataMatrix[r][c];
00184                     if (c < cols - 1) outFile << ",";
00185                 }
00186                 outFile << endl;
00187             }
00188 
00189             outFile.close();
00190             return true;
00191         }
00192     private:
00193         size_t rows;
00194         size_t cols;
00195         std::vector<std::string> colLabels;
00196         T** dataMatrix;
00198     };
00199 } // mdata
00200 
00201 #endif
00202 
00203 // ========================
00204 // End of datatable.h
00205 // ========================
```

6.3 include/experiment.h File Reference

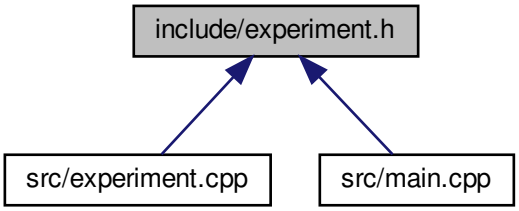
Header file for the Experiment class. Contains the basic logic and functions to run the cs471 project experiment.

```
#include <string>
#include <random>
#include <chrono>
#include <vector>
#include "mfunctions.h"
#include "inireader.h"
#include "population.h"
#include "threadpool.h"
#include "partswarm.h"
#include "firefly.h"
#include "harmsearch.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

Enumerations

- enum mfunc::Algorithm { mfunc::Algorithm::ParticleSwarm = 0, mfunc::Algorithm::Firefly = 1, mfunc::Algorithm::HarmonySearch = 2, mfunc::Algorithm::Count = 3 }

  Simple enum that selects one of the search algorithms.

6.3.1 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

0.4

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file experiment.h.

6.4 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 "partswarm.h"
00025 #include "firefly.h"
00026 #include "harmsearch.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
00044     enum class Algorithm
00045     {
00046         ParticleSwarm = 0,
00047         Firefly = 1,
00048         HarmonySearch = 2,
00049         Count = 3
00050     };
00051
00062     template<class T>
00063     class Experiment
00064     {
00065     public:
00066         Experiment();
00067         ~Experiment();
00068         bool init(const char* paramFile);
00069         int testAllFunc();
00070         int testPS();
00071         int testFA();
00072         int testHS();
00073     private:
00074         std::mutex popPoolMutex;
00075         util::IniReader iniParams;
00076         std::vector<mdata::Population<T>*> populationsPool;
00077         std::string resultsFile;
00078         std::string worstFitnessFile;
00079         std::string execTimesFile;
00080         std::string funcCallsFile;
00081         std::string populationsFile;
00082         RandomBounds<T>* vBounds;
00083         ThreadPool* tPool;
00084         size_t iterations;
00085         Algorithm selAlg;
00087         int runPSThreaded(PSParams<T> params, mdata::DataTable<T>* timesTable
      , size_t tRow, size_t tCol);
00088         int runFAThreaded(FAParams<T> params, mdata::DataTable<T>* timesTable
      , size_t tRow, size_t tCol);
00089         int runHSThreaded(HSParams<T> params, mdata::DataTable<T>* timesTable
      , size_t tRow, size_t tCol);
00090
00091         int waitThreadFutures(std::vector<std::future<int>>& futures);
00092
00093         const PSParams<T> createPSParamsTemplate();
00094         const FAParams<T> createFAParamsTemplate();
00095         const HSParams<T> createHSParamsTemplate();
00096
00097         mdata::Population<T>* popPoolRemove();
00098         void popPoolAdd(mdata::Population<T>* popPtr);
00099
00100         bool parseFuncBounds();
00101
00102         bool allocatePopulationPool(size_t count, size_t popSize, size_t dimensions);
00103         void releasePopulationPool();
00104
00105         bool allocateVBounds();
00106         void releaseVBounds();
00107
00108         bool allocateThreadPool(size_t numThreads);
00109         void releaseThreadPool();
00110     };
```

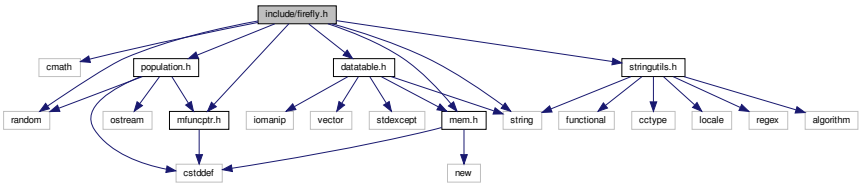
```
00111 } // mfunc
00112
00113 #endif
00114
00115 // ========================
00116 // End of experiment.h
00117 // ========================
```

6.5 include/firefly.h File Reference

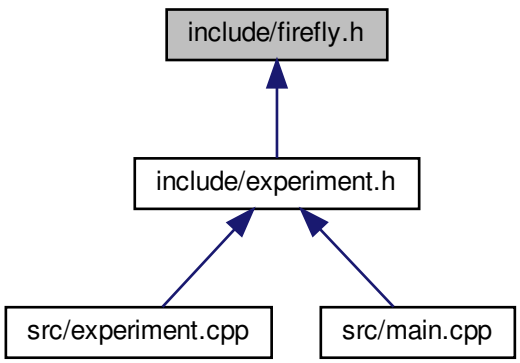
Contains the Firefly class, which runs the firefly algorithm using the given parameters.

```
#include <cmath>
#include <string>
#include "population.h"
#include "mfuncptr.h"
#include "datatable.h"
#include "random"
#include "mem.h"
#include "stringutils.h"
```

Include dependency graph for firefly.h:



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



Classes

- struct mfunc::FAParams< T >

  The FAParams struct contains various parameters that are required to be passed to the Firefly.run() method.
- class mfunc::Firefly< T >

  The Firefly class runs the firefly algorithm with the given parameters passed to the run() method.

Namespaces

- mfunc

Macros

- #define _USE_MATH_DEFINES
- #define BETA_INIT 1.0
- #define POPFILE_GEN_PATTERN "%GEN%"

6.5.1 Detailed Description

Contains the Firefly class, which runs the firefly algorithm using the given parameters.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-05-12

Copyright

Definition in file firefly.h.

6.5.2 Macro Definition Documentation

6.5.2.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

Definition at line 16 of file firefly.h.

6.5.2.2 BETA_INIT

```
#define BETA_INIT 1.0
```

Definition at line 27 of file firefly.h.

Referenced by mfunc::Firefly< T >::run().

6.5.2.3 POPFILE_GEN_PATTERN

```
#define POPFILE_GEN_PATTERN "%GEN%"
```

Definition at line 28 of file firefly.h.

Referenced by mfunc::Firefly< T >::run().

6.6 firefly.h

```
00001
00013 #ifndef __FIREFLY_H
00014 #define __FIREFLY_H
00015
00016 #define _USE_MATH_DEFINES
00017
00018 #include <cmath>
00019 #include <string>
00020 #include "population.h"
00021 #include "mfuncptr.h"
00022 #include "datatable.h"
00023 #include "random"
00024 #include "mem.h"
00025 #include "stringutils.h"
00026
00027 #define BETA_INIT 1.0
00028 #define POPFILE_GEN_PATTERN "%GEN%"
00029
00030 namespace mfunc
00031 {
00038     template <class T>
00039     struct FAParams
00040     {
00041         std::string popFile; // String file name for population dump file
00042         mdata::DataTable<T>* bestFitnessTable; // Data table for best
      fitness values
00043         mdata::DataTable<T>* worstFitnessTable; // Data table for worst
      fitness values
00044         size_t fitTableCol; // Data table column for best and worst fitness values
00045         mdata::Population<T>* mainPop; // Pointer to main population object
00046         mdata::Population<T>* nextPop; // Pointer to next population object
00047         mfuncPtr<T> fPtr; // Function pointer to the objective function being tested
00048         T fMinBound; // Minimum population vector bounds for objective function
00049         T fMaxBound; // Maximum population vector bounds for objective function
00050         unsigned int iterations; // Number of iterations to run search algorithm
00051         double alpha; // Alpha parameter for firefly algorithm
```

```
00052         double betamin; // Betamin parameter for firefly algorithm
00053         double gamma; // Gamma parameter for firefly algorithm
00054
00058         FAParams()
00059         {
00060             popFile = "";
00061             bestFitnessTable = nullptr;
00062             worstFitnessTable = nullptr;
00063             fitTableCol = 0;
00064             mainPop = nullptr;
00065             nextPop = nullptr;
00066             fPtr = nullptr;
00067             fMinBound = 0;
00068             fMaxBound = 0;
00069             iterations = 0;
00070             alpha = 0;
00071             betamin = 0;
00072             gamma = 0;
00073         }
00074     };
00075
00082     template <class T>
00083     class Firefly
00084     {
00085     public:
00086         Firefly();
00087         ~Firefly() = default;
00088         int run(FAParams<T> p);
00089     private:
00090         std::random_device seed;
00091         std::mt19937 engine;
00092         std::uniform_real_distribution<T> rchance;
00093
00094         void evaluate(FAParams<T>& p, T* solBuffer, size_t firefly);
00095         void move(FAParams<T>& p, T* solBuffer, size_t firefly_i, size_t firefly_j);
00096         T calcDistance(T* fv_i, T* fv_j, size_t dimSize);
00097     };
00098 }
00099
00105 template <class T>
00106 mfunc::Firefly<T>::Firefly()
00107     : seed(), engine(seed()), rchance(0, 1)
00108 {
00109 }
00110
00118 template <class T>
00119 int mfunc::Firefly<T>::run(FAParams<T> p)
00120 {
00121     if (p.mainPop == nullptr || p.nextPop == nullptr || p.fPtr == nullptr)
00122         return 1;
00123
00124     // Get population information
00125     const size_t popSize = p.mainPop->getPopulationSize();
00126     const size_t dimSize = p.mainPop->getDimensionsSize();
00127
00128     T* solBuffer = util::allocArray<T>(dimSize);
00129     if (solBuffer == nullptr)
00130         return 2;
00131
00132     // Generate population vectors
00133     if (!p.nextPop->generate(p.fMinBound, p.fMaxBound))
00134         return 3;
00135
00136     // Calculate fitness for all population vectors
00137     if (!p.nextPop->calcAllFitness(p.fPtr))
00138         return 4;
00139
00140     // Sort population from worst to best
00141     p.nextPop->sortFitnessDescend();
00142
00143     for (unsigned int iter = 0; iter < p.iterations; iter++)
00144     {
00145         p.mainPop->copyAllFrom(p.nextPop);
00146
00147         for (size_t firefly_i = 0; firefly_i < popSize; firefly_i++)
00148         {
00149             evaluate(p, solBuffer, firefly_i);
00150         }
00151
00152         p.nextPop->sortFitnessDescend();
00153
00154         // Store best fitness for this iteration
00155         if (p.bestFitnessTable != nullptr)
00156             p.bestFitnessTable->setEntry(iter, p.fitTableCol, p.
      nextPop->getFitness(popSize - 1));
00157
00158         // Store worst fitness for this iteration
```

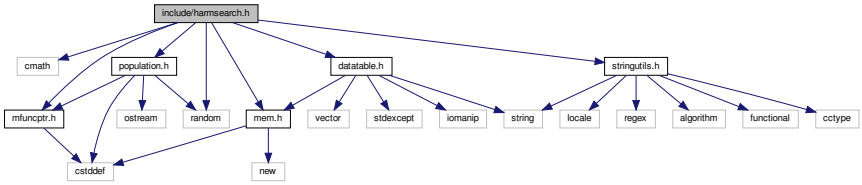
```
00159         if (p.worstFitnessTable != nullptr)
00160             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.
      nextPop->getFitness(0));
00161 
00162         // Dump population vectors to file
00163         if (!p.popFile.empty())
00164             p.nextPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
      POPFILE_GEN_PATTERN), std::to_string(iter)));
00165     }
00166 
00167     util::releaseArray(solBuffer);
00168 
00169     return 0;
00170 }
00171 
00180 template <class T>
00181 void mfunc::Firefly<T>::evaluate(FAParams<T>& p, T* solBuffer, size_t
       firefly_i)
00182 {
00183     const size_t popSize = p.mainPop->getPopulationSize();
00184 
00185     // Compare every other firefly with firefly_i, and move it
00186     // towards firefly_i if fitness is worse
00187     for (size_t firefly_j = 0; firefly_j < popSize; firefly_j++)
00188     {
00189         const T light_j = p.mainPop->getFitness(firefly_j);
00190 
00191         if (p.nextPop->getFitness(firefly_i) < light_j)
00192         {
00193             move(p, solBuffer, firefly_j, firefly_i);
00194         }
00195     }
00196 }
00197 
00208 template <class T>
00209 void mfunc::Firefly<T>::move(FAParams<T>& p, T* solBuffer, size_t
      firefly_j, size_t firefly_i)
00210 {
00211     const size_t dimSize = p.mainPop->getDimensionsSize();
00212 
00213     auto fv_j = p.mainPop->getPopulationPtr(firefly_j);
00214     auto fv_i_next = p.nextPop->getPopulationPtr(firefly_i);
00215 
00216     // Calculate distance between the two fireflies and then their beta value
00217     T r = calcDistance(fv_i_next, fv_j, dimSize);
00218     T betaDist = std::pow(static_cast<T>(M_E), -1 * p.gamma * r);
00219     T beta = (BETA_INIT - p.betamin) * betaDist + p.betamin;
00220 
00221     for (size_t d = 0; d < dimSize; d++)
00222     {
00223         // Calculate new value for current dimension
00224         T alpha = p.alpha * (rchance(engine) - 0.5) * (std::abs(p.fMaxBound - p.
      fMinBound));
00225         solBuffer[d] = fv_j[d] + (beta * (fv_i_next[d] - fv_j[d])) + alpha;
00226 
00227         if (solBuffer[d] < p.fMinBound)
00228             solBuffer[d] = p.fMinBound;
00229         else if (solBuffer[d] > p.fMaxBound)
00230             solBuffer[d] = p.fMaxBound;
00231     }
00232 
00233     // Calculate fitness for new firefly
00234     T newFit = p.fPtr(solBuffer, dimSize);
00235     T oldFit = p.nextPop->getFitness(firefly_j);
00236 
00237     // Update firefly if new is better than old
00238     if (newFit < oldFit);
00239     {
00240         p.nextPop->copyPopulation(solBuffer, firefly_j);
00241         p.nextPop->setFitness(firefly_j, newFit);
00242     }
00243 }
00244 
00254 template <class T>
00255 T mfunc::Firefly<T>::calcDistance(T* fv_i, T* fv_j, size_t dimSize)
00256 {
00257     T sum = 0;
00258     for (size_t d = 0; d < dimSize; d++)
00259     {
00260         T diff = fv_i[d] - fv_j[d];
00261         sum += diff * diff;
00262     }
00263 
00264     return std::sqrt(sum);
00265 }
00266 
00267 #endif
```

6.7 include/harmsearch.h File Reference

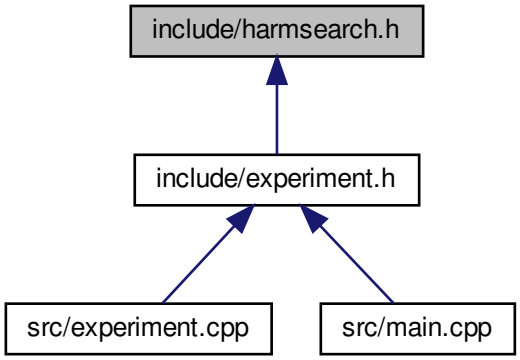
Contains the HarmonySearch class, which runs the harmony search algorithm using the given parameters.

```
#include <cmath>
#include "population.h"
#include "mfuncptr.h"
#include "datatable.h"
#include "random"
#include "mem.h"
#include "stringutils.h"
```

Include dependency graph for harmsearch.h:



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



Classes

- struct mfunc::HSParams< T >

  The HSParams struct contains various parameters that are required to be passed to the HarmonySearch.run() method.
- class mfunc::HarmonySearch< T >

  The HarmonySearch class runs the harmony search algorithm based on the parameters passed to the run() method.

Namespaces

- mfunc

Macros

- #define POPFILE_GEN_PATTERN "%GEN%"

6.7.1 Detailed Description

Contains the HarmonySearch class, which runs the harmony search algorithm using the given parameters.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-05-13

Copyright

Copyright (c) 2019

Definition in file harmsearch.h.

6.7.2 Macro Definition Documentation

6.7.2.1 POPFILE_GEN_PATTERN

```
#define POPFILE_GEN_PATTERN "%GEN%"
```

Definition at line 24 of file harmsearch.h.

Referenced by mfunc::HarmonySearch< T >::run().

6.8 harmsearch.h

```
00001
00013 #ifndef __HARMSEARCH_H
00014 #define __HARMSEARCH_H
00015
00016 #include <cmath>
00017 #include "population.h"
00018 #include "mfuncptr.h"
00019 #include "datatable.h"
00020 #include "random"
00021 #include "mem.h"
00022 #include "stringutils.h"
00023
00024 #define POPFILE_GEN_PATTERN "%GEN%"
00025
00026 namespace mfunc
00027 {
00034     template <class T>
00035     struct HSParams
00036     {
00037         std::string popFile; // String file name for population dump file
00038         mdata::DataTable<T>* bestFitnessTable; // Data table for best
      fitness values
00039         mdata::DataTable<T>* worstFitnessTable; // Data table for worst
      fitness values
00040         size_t fitTableCol; // Data table column for best and worst fitness values
00041         mdata::Population<T>* mainPop; // Pointer to main population object
00042         mfuncPtr<T> fPtr; // Function pointer to the objective function being tested
00043         T fMinBound; // Minimum population vector bounds for objective function
00044         T fMaxBound; // Maximum population vector bounds for objective function
00045         unsigned int iterations; // Number of iterations to run search algorithm
00046         double hmcr; // HMCR parameter for harmony search
00047         double par; // PAR parameter for harmony search
00048         double bw; // BW parameter for harmony search
00049
00053         HSParams()
00054         {
00055             popFile = "";
00056             bestFitnessTable = nullptr;
00057             worstFitnessTable = nullptr;
00058             fitTableCol = 0;
00059             mainPop = nullptr;
00060             fPtr = nullptr;
00061             fMinBound = 0;
00062             fMaxBound = 0;
00063             iterations = 0;
00064             hmcr = 0;
00065             par = 0;
00066             bw = 0;
00067         }
00068     };
00069
00076     template <class T>
00077     class HarmonySearch
00078     {
00079     public:
00080         HarmonySearch();
00081         ~HarmonySearch() = default;
00082         int run(HSParams<T> p);
00083     private:
00084         std::random_device seed;
00085         std::mt19937 engine;
00086         std::uniform_real_distribution<T> rchance;
00087         std::uniform_real_distribution<T> rrange;
00088
00089         void adjustPitch(HSParams<T>& p, T* solBuffer, const size_t numDim);
00090     };
00091 } // namespace mfunc
00092
00098 template <class T>
00099 mfunc::HarmonySearch<T>::HarmonySearch()
00100     : seed(), engine(seed()), rchance(0, 1), rrange(-1, 1)
00101 {
00102 }
00103
00111 template <class T>
00112 int mfunc::HarmonySearch<T>::run(HSParams<T> p)
00113 {
00114     if (p.mainPop == nullptr || p.fPtr == nullptr)
00115         return 1;
00116
00117     // Get population information
00118     const size_t popSize = p.mainPop->getPopulationSize();
00119     const size_t dimSize = p.mainPop->getDimensionsSize();
00120
```

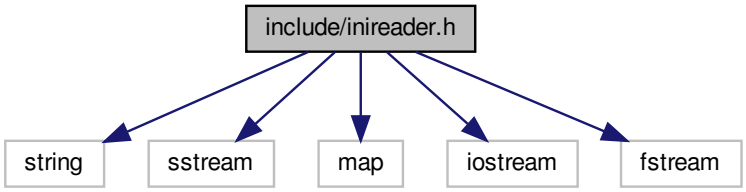
```
00121     T* solBuffer = util::allocArray<T>(dimSize);
00122     if (solBuffer == nullptr)
00123         return 2;
00124 
00125     // Generate random population vectors
00126     if (!p.mainPop->generate(p.fMinBound, p.fMaxBound))
00127         return 3;
00128 
00129     // Calculate fitness values for entire population
00130     if (!p.mainPop->calcAllFitness(p.fPtr))
00131         return 4;
00132 
00133     // Sort fitness from best to worst
00134     p.mainPop->sortFitnessAscend();
00135 
00136     for (unsigned int iter = 0; iter < p.iterations; iter++)
00137     {
00138         // Generate new solution
00139         adjustPitch(p, solBuffer, dimSize);
00140 
00141         // Calculate the new fitness, and replace worst if new solution is better
00142         T newAesthetic = p.fPtr(solBuffer, dimSize);
00143         T oldAesthetic = p.mainPop->getFitness(popSize - 1);
00144         if (newAesthetic < oldAesthetic)
00145         {
00146             p.mainPop->copyPopulation(solBuffer, popSize - 1);
00147             p.mainPop->setFitness(popSize - 1, newAesthetic);
00148         }
00149 
00150         // Resort population
00151         p.mainPop->sortFitnessAscend();
00152 
00153         // Store best fitness value for this iteration
00154         if (p.bestFitnessTable != nullptr)
00155             p.bestFitnessTable->setEntry(iter, p.fitTableCol, p.
      mainPop->getFitness(0));
00156 
00157         // Store worst fitness value for this iteration
00158         if (p.worstFitnessTable != nullptr)
00159             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.
      mainPop->getFitness(popSize - 1));
00160 
00161         // Dump population vectors to a file
00162         if (!p.popFile.empty())
00163             p.mainPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
      POPFILE_GEN_PATTERN), std::to_string(iter)));
00164     }
00165 
00166     util::releaseArray<T>(solBuffer);
00167 
00168     return 0;
00169 }
00170 
00180 template <class T>
00181 void mfunc::HarmonySearch<T>::adjustPitch(
      HSParams<T>& p, T* solBuffer, const size_t numDim)
00182 {
00183     // Set up random number distribution for a random population vector
00184     const size_t popSize = p.mainPop->getPopulationSize();
00185     auto randPop = std::uniform_int_distribution<size_t>(0, popSize - 1);
00186 
00187     for (size_t dim = 0; dim < numDim; dim++)
00188     {
00189         T newPitch = 0;
00190         if (rchance(engine) <= p.hmcr)
00191         {
00192             // Get random value from existing population
00193             newPitch = p.mainPop->getPopulationPtr(randPop(engine))[dim];
00194             if (rchance(engine) <= p.par)
00195             {
00196                 // Adjust pitch of selected value
00197                 newPitch += rrange(engine) * p.bw;
00198             }
00199         }
00200         else
00201         {
00202             // Generate a new completely random value for this dimension
00203             newPitch = (rchance(engine) - 0.5) * std::abs(p.fMaxBound - p.
      fMinBound);
00204         }
00205 
00206         solBuffer[dim] = newPitch;
00207     }
00208 }
00209 
00210 
00211 #endif
```

6.9 include/inireader.h File Reference

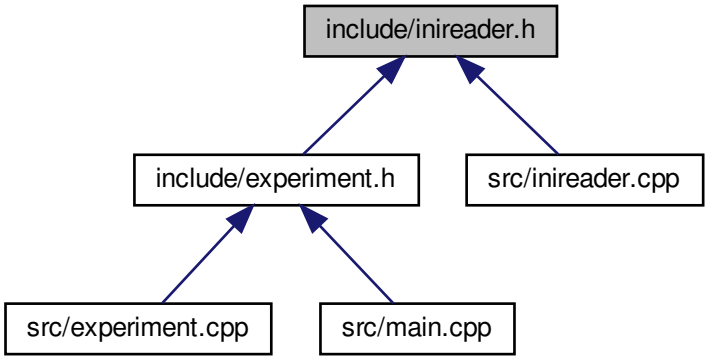
Header file for the IniReader class, which can open and parse simple ∗.ini files.

```
#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

6.9.1 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.

6.10 inireader.h

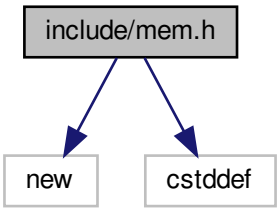
```
00001
00013 #ifndef __INIREADER_H
00014 #define __INIREADER_H
00015
00016 #include <string>
00017 #include <sstream>
00018 #include <map>
00019 #include <iostream>
00020 #include <fstream>
00021
00022 namespace util
00023 {
00046     class IniReader
00047     {
00048     public:
00049         IniReader();
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, std::string defVal = "");
00055
00056         template <class T>
00057         T getEntryAs(std::string section, std::string entry, T defVal = {})
00058         {
00059             std::stringstream ss(getEntry(section, entry, std::to_string(defVal)));
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;
00068         bool parseFile();
00069         void parseEntry(const std::string& sectionName, const std::string& entry);
00070     };
00071 }
00072
00073 #endif
00074
00075 // ========================
00076 // End of inireader.h
00077 // ========================
```

6.11 include/mem.h File Reference

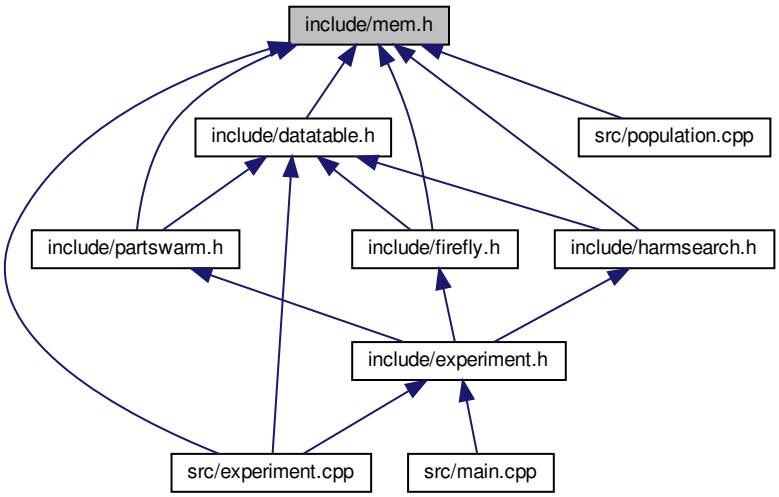
Header file for various memory utility functions.

```
#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>
  bool 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.

6.11.1 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.

6.12 mem.h

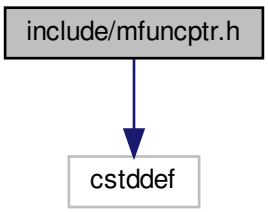
```
00001
00012 #ifndef __MEM_H
00013 #define __MEM_H
00014
00015 #include <new> // std::nothrow
00016 #include <cstddef> // size_t definition
00017
00018 namespace util
00019 {
00028     template <class T = double>
00029     inline void initArray(T* a, size_t size, T val)
00030     {
00031         if (a == nullptr) return;
00032
00033         for (size_t i = 0; i < size; i++)
00034         {
00035             a[i] = val;
00036         }
00037     }
00038
00048     template <class T = double>
00049     inline void initMatrix(T** m, size_t rows, size_t cols, T val)
00050     {
00051         if (m == nullptr) return;
00052
00053         for (size_t i = 0; i < rows; i++)
00054         {
00055             initArray(m[i], cols, val);
00056         }
00057     }
00058
00065     template <class T = double>
00066     bool releaseArray(T*& a)
00067     {
00068         if (a == nullptr) return true;
00069
00070         try
00071         {
00072             delete[] a;
00073             a = nullptr;
00074             return true;
00075         }
00076         catch(...)
00077         {
00078             return false;
00079         }
00080     }
00081
00089     template <class T = double>
00090     void releaseMatrix(T**& m, size_t rows)
00091     {
00092         if (m == nullptr) return;
00093
00094         for (size_t i = 0; i < rows; i++)
00095         {
00096             if (m[i] != nullptr)
00097             {
00098                 // Release each row
00099                 releaseArray<T>(m[i]);
00100             }
00101         }
00102
00103         // Release columns
00104         delete[] m;
00105         m = nullptr;
00106     }
00107
00115     template <class T = double>
00116     inline T* allocArray(size_t size)
00117     {
00118         return new(std::nothrow) T[size];
00119     }
00120
00129     template <class T = double>
00130     inline T** allocMatrix(size_t rows, size_t cols)
00131     {
00132         T** m = (T**)allocArray<T*>(rows);
00133         if (m == nullptr) return nullptr;
00134
00135         for (size_t i = 0; i < rows; i++)
00136         {
00137             m[i] = allocArray<T>(cols);
00138             if (m[i] == nullptr)
00139             {
```

```
00140                 releaseMatrix<T>(m, rows);
00141                 return nullptr;
00142             }
00143         }
00144 
00145         return m;
00146     }
00147 
00156     template <class T = double>
00157     inline void copyArray(T* src, T* dest, size_t size)
00158     {
00159         for (size_t i = 0; i < size; i++)
00160             dest[i] = src[i];
00161     }
00162 }
00163 
00164 #endif
00165 
00166 // ==========================
00167 // End of mem.h
00168 // ==========================
```

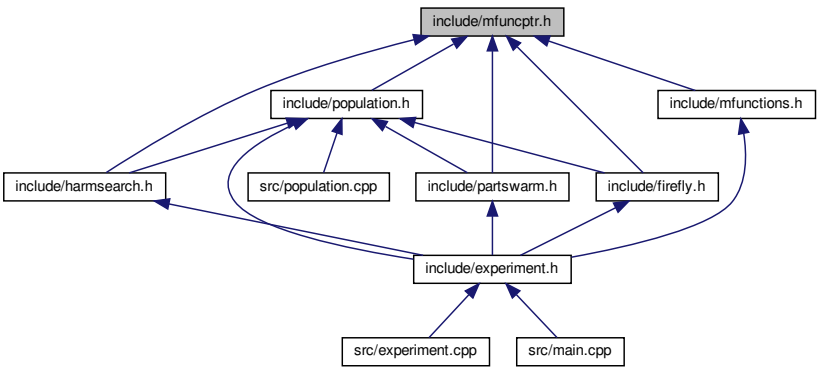
6.13 include/mfuncptr.h File Reference

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.*

6.13.1 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.

6.14 mfuncptr.h

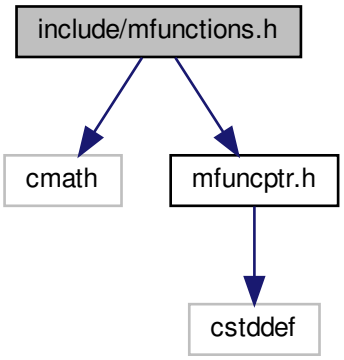
```
00001
00014 #ifndef __MFUNCPTR_H
00015 #define __MFUNCPTR_H
00016
00017 #include <cstddef> // size_t definition
00018
00019 namespace mfunc
00020 {
00027     template <class T>
00028     using mfuncPtr = T (*)(T*, size_t);
00029 }
00030
00031 #endif
00032
00033 // =========================
00034 // End of mfuncptr.h
00035 // =========================
```

6.15 include/mfunctions.h File Reference

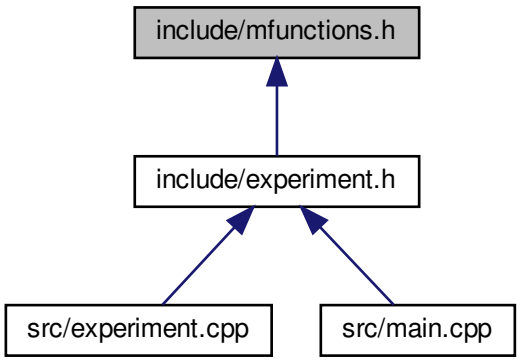
Contains various math function definitions.

```
#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 _NUM_FUNCTIONS 18
- #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"
- #define _schwefelId 1
- #define _dejongId 2
- #define _rosenbrokId 3
- #define _rastriginId 4
- #define _griewangkId 5
- #define _sineEnvelopeSineWaveId 6
- #define _stretchedVSineWaveId 7
- #define _ackleysOneId 8
- #define _ackleysTwoId 9
- #define _eggHolderId 10
- #define _ranaId 11
- #define _pathologicalId 12
- #define _michalewiczId 13
- #define _mastersCosineWaveId 14
- #define _quarticId 15
- #define _levyId 16
- #define _stepId 17
- #define _alpineId 18

Variables

- constexpr const unsigned int mfunc::NUM_FUNCTIONS = _NUM_FUNCTIONS

6.15.1 Detailed Description

Contains various math function definitions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-03-29

Copyright

Definition in file mfunctions.h.

6.15.2 Macro Definition Documentation

6.15.2.1 _ackleysOneDesc

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

Definition at line 29 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.2 _ackleysOneId

```
#define _ackleysOneId 8
```

Definition at line 48 of file mfunctions.h.

Referenced by mfunc::Functions< T >::ackleysOne(), mfunc::FunctionDesc::get(), and mfunc::Functions< T >::get().

6.15.2.3 _ackleysTwoDesc

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

Definition at line 30 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.4 _ackleysTwoId

```
#define _ackleysTwoId 9
```

Definition at line 49 of file mfunctions.h.

Referenced by mfunc::Functions< T >::ackleysTwo(), mfunc::FunctionDesc::get(), and mfunc::Functions< T >::get().

6.15.2.5 _alpineDesc

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

Definition at line 39 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.6 _alpineId

```
#define _alpineId 18
```

Definition at line 58 of file mfunctions.h.

Referenced by mfunc::Functions< T >::alpine(), mfunc::FunctionDesc::get(), and mfunc::Functions< T >::get().

6.15.2.7 _dejongDesc

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

Definition at line 23 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.8 _dejongId

```
#define _dejongId 2
```

Definition at line 42 of file mfunctions.h.

Referenced by mfunc::Functions< T >::dejong(), mfunc::FunctionDesc::get(), and mfunc::Functions< T >::get().

6.15.2.9 _eggHolderDesc

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

Definition at line 31 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.10 _eggHolderId

```
#define _eggHolderId 10
```

Definition at line 50 of file mfunctions.h.

Referenced by mfunc::Functions< T >::eggHolder(), mfunc::FunctionDesc::get(), and mfunc::Functions< T >::get().

6.15.2.11 _griewangkDesc

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

Definition at line 26 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.12 _griewangkId

```
#define _griewangkId 5
```

Definition at line 45 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::griewangk().

6.15.2.13 _levyDesc

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

Definition at line 37 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.14 _levyId

```
#define _levyId 16
```

Definition at line 56 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::levy().

6.15.2.15 _mastersCosineWaveDesc

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

Definition at line 35 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.16 _mastersCosineWaveId

```
#define _mastersCosineWaveId 14
```

Definition at line 54 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::masters↩ CosineWave().

6.15.2.17 _michalewiczDesc

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

Definition at line 34 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.18 _michalewiczId

```
#define _michalewiczId 13
```

Definition at line 53 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::michalewicz().

6.15.2.19 _NUM_FUNCTIONS

```
#define _NUM_FUNCTIONS 18
```

Definition at line 20 of file mfunctions.h.

Referenced by mfunc::Functions< T >::getCallCounter(), and mfunc::Functions< T >::resetCallCounters().

6.15.2.20 _pathologicalDesc

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

Definition at line 33 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.21 _pathologicalId

```
#define _pathologicalId 12
```

Definition at line 52 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::pathological().

6.15.2.22 _quarticDesc

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

Definition at line 36 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.23 _quarticId

```
#define _quarticId 15
```

Definition at line 55 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::quartic().

6.15.2.24 _ranaDesc

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

Definition at line 32 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.25 _ranaId

```
#define _ranaId 11
```

Definition at line 51 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::rana().

6.15.2.26 _rastriginDesc

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

Definition at line 25 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.27 _rastriginId

```
#define _rastriginId 4
```

Definition at line 44 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::rastrigin().

6.15.2.28 _rosenbrokDesc

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

Definition at line 24 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.29 _rosenbrokId

```
#define _rosenbrokId 3
```

Definition at line 43 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::rosenbrok().

6.15.2.30 _schwefelDesc

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

Definition at line 22 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.31 _schwefelId

```
#define _schwefelId 1
```

Definition at line 41 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::schwefel().

6.15.2.32 _sineEnvelopeSineWaveDesc

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

Definition at line 27 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.33 _sineEnvelopeSineWaveId

```
#define _sineEnvelopeSineWaveId 6
```

Definition at line 46 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::sineEnvelopeSineWave().

6.15.2.34 _stepDesc

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

Definition at line 38 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.35 _stepId

```
#define _stepId 17
```

Definition at line 57 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::step().

6.15.2.36 _stretchedVSineWaveDesc

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

Definition at line 28 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get().

6.15.2.37 _stretchedVSineWaveId

```
#define _stretchedVSineWaveId 7
```

Definition at line 47 of file mfunctions.h.

Referenced by mfunc::FunctionDesc::get(), mfunc::Functions< T >::get(), and mfunc::Functions< T >::stretchedVSineWave().

6.15.2.38 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

Definition at line 15 of file mfunctions.h.

6.16 mfunctions.h

```
00001
00012 #ifndef __MFUNCIONS_H
00013 #define __MFUNCIONS_H
00014
00015 #define _USE_MATH_DEFINES
00016
00017 #include <cmath>
00018 #include "mfuncptr.h"
00019
00020 #define _NUM_FUNCTIONS 18
00021
00022 #define _schwefelDesc "Schwefel's function"
00023 #define _dejongDesc "1st De Jong's function"
00024 #define _rosenbrokDesc "Rosenbrock"
00025 #define _rastriginDesc "Rastrigin"
00026 #define _griewangkDesc "Griewangk"
00027 #define _sineEnvelopeSineWaveDesc "Sine Envelope Sine Wave"
00028 #define _stretchedVSineWaveDesc "Stretched V Sine Wave"
00029 #define _ackleysOneDesc "Ackley's One"
00030 #define _ackleysTwoDesc "Ackley's Two"
00031 #define _eggHolderDesc "Egg Holder"
00032 #define _ranaDesc "Rana"
00033 #define _pathologicalDesc "Pathological"
00034 #define _michalewiczDesc "Michalewicz"
00035 #define _mastersCosineWaveDesc "Masters Cosine Wave"
00036 #define _quarticDesc "Quartic"
00037 #define _levyDesc "Levy"
00038 #define _stepDesc "Step"
00039 #define _alpineDesc "Alpine"
00040
00041 #define _schwefelId 1
00042 #define _dejongId 2
00043 #define _rosenbrokId 3
00044 #define _rastriginId 4
00045 #define _griewangkId 5
00046 #define _sineEnvelopeSineWaveId 6
00047 #define _stretchedVSineWaveId 7
00048 #define _ackleysOneId 8
00049 #define _ackleysTwoId 9
00050 #define _eggHolderId 10
00051 #define _ranaId 11
00052 #define _pathologicalId 12
00053 #define _michalewiczId 13
00054 #define _mastersCosineWaveId 14
00055 #define _quarticId 15
00056 #define _levyId 16
00057 #define _stepId 17
00058 #define _alpineId 18
00059
00062 namespace mfunc
00063 {
00067     constexpr const unsigned int NUM_FUNCTIONS = _NUM_FUNCTIONS;
00068
00076     struct FunctionDesc
00077     {
00078         static const char* get(unsigned int f)
00079         {
00080             switch (f)
00081             {
00082                 case _schwefelId:
00083                     return _schwefelDesc;
00084                 case _dejongId:
00085                     return _dejongDesc;
00086                 case _rosenbrokId:
00087                     return _rosenbrokDesc;
00088                 case _rastriginId:
00089                     return _rastriginDesc;
00090                 case _griewangkId:
00091                     return _griewangkDesc;
00092                 case _sineEnvelopeSineWaveId:
00093                     return _sineEnvelopeSineWaveDesc;
```

```
00094                 case _stretchedVSineWaveId:
00095                     return _stretchedVSineWaveDesc;
00096                 case _ackleysOneId:
00097                     return _ackleysOneDesc;
00098                 case _ackleysTwoId:
00099                     return _ackleysTwoDesc;
00100                 case _eggHolderId:
00101                     return _eggHolderDesc;
00102                 case _ranaId:
00103                     return _ranaDesc;
00104                 case _pathologicalId:
00105                     return _pathologicalDesc;
00106                 case _michalewiczId:
00107                     return _michalewiczDesc;
00108                 case _mastersCosineWaveId:
00109                     return _mastersCosineWaveDesc;
00110                 case _quarticId:
00111                     return _quarticDesc;
00112                 case _levyId:
00113                     return _levyDesc;
00114                 case _stepId:
00115                     return _stepDesc;
00116                 case _alpineId:
00117                     return _alpineDesc;
00118                 default:
00119                     return NULL;
00120             }
00121         }
00122     };
00123
00131     template <class T>
00132     struct Functions
00133     {
00134         static T schwefel(T* v, size_t n);
00135         static T dejong(T* v, size_t n);
00136         static T rosenbrok(T* v, size_t n);
00137         static T rastrigin(T* v, size_t n);
00138         static T griewangk(T* v, size_t n);
00139         static T sineEnvelopeSineWave(T* v, size_t n);
00140         static T stretchedVSineWave(T* v, size_t n);
00141         static T ackleysOne(T* v, size_t n);
00142         static T ackleysTwo(T* v, size_t n);
00143         static T eggHolder(T* v, size_t n);
00144         static T rana(T* v, size_t n);
00145         static T pathological(T* v, size_t n);
00146         static T mastersCosineWave(T* v, size_t n);
00147         static T michalewicz(T* v, size_t n);
00148         static T quartic(T* v, size_t n);
00149         static T levy(T* v, size_t n);
00150         static T step(T* v, size_t n);
00151         static T alpine(T* v, size_t n);
00152         static mfuncPtr<T> get(unsigned int f);
00153         static bool exec(unsigned int f, T* v, size_t n, T& outResult);
00154         static T nthroot(T x, T n);
00155         static T w(T x);
00156         static size_t getCallCounter(unsigned int f);
00157         static void resetCallCounters();
00158     private:
00159         static size_t fCallCounters[_NUM_FUNCTIONS];
00160         static bool fCountersInit;
00161
00162         static void fCounterInc(unsigned int f);
00163     };
00164 }
00165
00169 template <class T>
00170 bool mfunc::Functions<T>::fCountersInit = false;
00171
00176 template <class T>
00177 size_t mfunc::Functions<T>::fCallCounters[
      _NUM_FUNCTIONS];
00178
00185 template <class T>
00186 T mfunc::Functions<T>::nthroot(T x, T n)
00187 {
00188     return std::pow(x, static_cast<T>(1.0) / n);
00189 }
00190
00191 // ============================================
00192
00200 template <class T>
00201 T mfunc::Functions<T>::schwefel(T* v, size_t n)
00202 {
00203     fCounterInc(_schwefelId);
00204
00205     T f = 0.0;
00206
```

```
00207     for (size_t i = 0; i < n; i++)
00208     {
00209         f += (static_cast<T>(-1.0) * v[i]) * std::sin(std::sqrt(std::abs(v[i])));
00210     }
00211 
00212     return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
00213 }
00214 
00215 // ================================================
00216 
00224 template <class T>
00225 T mfunc::Functions<T>::dejong(T* v, size_t n)
00226 {
00227     fCounterInc(_dejongId);
00228 
00229     T f = 0.0;
00230 
00231     for (size_t i = 0; i < n; i++)
00232     {
00233         f += v[i] * v[i];
00234     }
00235 
00236     return f;
00237 }
00238 
00239 // ================================================
00240 
00248 template <class T>
00249 T mfunc::Functions<T>::rosenbrok(T* v, size_t n)
00250 {
00251     fCounterInc(_rosenbrokId);
00252 
00253     T f = 0.0;
00254 
00255     for (size_t i = 0; i < n - 1; i++)
00256     {
00257         T a = ((v[i] * v[i]) - v[i+1]);
00258         T b = (static_cast<T>(1.0) - v[i]);
00259         f += static_cast<T>(100.0) * a * a;
00260         f += b * b;
00261     }
00262 
00263     return f;
00264 }
00265 
00266 // ================================================
00267 
00275 template <class T>
00276 T mfunc::Functions<T>::rastrigin(T* v, size_t n)
00277 {
00278     fCounterInc(_rastriginId);
00279 
00280     T f = 0.0;
00281 
00282     for (size_t i = 0; i < n; i++)
00283     {
00284         f += (v[i] * v[i]) - (static_cast<T>(10.0) * std::cos(static_cast<T>(2.0) * static_cast<T>(M_PI) *
      v[i]));
00285     }
00286 
00287     return static_cast<T>(10.0) * static_cast<T>(n) * f;
00288 }
00289 
00290 // ================================================
00291 
00299 template <class T>
00300 T mfunc::Functions<T>::griewangk(T* v, size_t n)
00301 {
00302     fCounterInc(_griewangkId);
00303 
00304     T sum = 0.0;
00305     T product = 0.0;
00306 
00307     for (size_t i = 0; i < n; i++)
00308     {
00309         sum += (v[i] * v[i]) / static_cast<T>(4000.0);
00310     }
00311 
00312     for (size_t i = 0; i < n; i++)
00313     {
00314         product *= std::cos(v[i] / std::sqrt(static_cast<T>(i + 1.0)));
00315     }
00316 
00317     return static_cast<T>(1.0) + sum - product;
00318 }
00319 
00320 // ================================================
```

```
00321
00329 template <class T>
00330 T mfunc::Functions<T>::sineEnvelopeSineWave(T* v, size_t n)
00331 {
00332     fCounterInc(_sineEnvelopeSineWaveId);
00333
00334     T f = 0.0;
00335
00336     for (size_t i = 0; i < n - 1; i++)
00337     {
00338         T a = std::sin(v[i]*v[i] + v[i+1]*v[i+1] - static_cast<T>(0.5));
00339         a *= a;
00340         T b = (static_cast<T>(1.0) + static_cast<T>(0.001)*(v[i]*v[i] + v[i+1]*v[i+1]));
00341         b *= b;
00342         f += static_cast<T>(0.5) + (a / b);
00343     }
00344
00345     return static_cast<T>(-1.0) * f;
00346 }
00347
00348 // ================================================
00349
00357 template <class T>
00358 T mfunc::Functions<T>::stretchedVSineWave(T* v, size_t n)
00359 {
00360     fCounterInc(_stretchedVSineWaveId);
00361
00362     T f = 0.0;
00363
00364     for (size_t i = 0; i < n - 1; i++)
00365     {
00366         T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
00367         T b = std::sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0)));
00368         b *= b;
00369         f += a * b + static_cast<T>(1.0);
00370     }
00371
00372     return f;
00373 }
00374
00375 // ================================================
00376
00384 template <class T>
00385 T mfunc::Functions<T>::ackleysOne(T* v, size_t n)
00386 {
00387     fCounterInc(_ackleysOneId);
00388
00389     T f = 0.0;
00390
00391     for (size_t i = 0; i < n - 1; i++)
00392     {
00393         T a = (static_cast<T>(1.0) / std::pow(static_cast<T>(M_E), static_cast<T>(0.2))) * std::sqrt(v[i]*v
    [i] + v[i+1]*v[i+1]);
00394         T b = static_cast<T>(3.0) * (std::cos(static_cast<T>(2.0) * v[i]) + std::sin(static_cast<T>(2.0) *
    v[i+1]));
00395         f += a + b;
00396     }
00397
00398     return f;
00399 }
00400
00401 // ================================================
00402
00410 template <class T>
00411 T mfunc::Functions<T>::ackleysTwo(T* v, size_t n)
00412 {
00413     fCounterInc(_ackleysTwoId);
00414
00415     T f = 0.0;
00416
00417     for (size_t i = 0; i < n - 1; i++)
00418     {
00419         T a = static_cast<T>(20.0) / std::pow(static_cast<T>(M_E), static_cast<T>(0.2) * std::sqrt((v[i]*v[
    i] + v[i+1]*v[i+1]) / static_cast<T>(2.0)));
00420         T b = std::pow(static_cast<T>(M_E), static_cast<T>(0.5) *
00421             (std::cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + std::cos(static_cast<T>(2.0) *
    static_cast<T>(M_PI) * v[i+1])));
00422         f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
00423     }
00424
00425     return f;
00426 }
00427
00428 // ================================================
00429
00437 template <class T>
00438 T mfunc::Functions<T>::eggHolder(T* v, size_t n)
```

```
00439 {
00440     fCounterInc(_eggHolderId);
00441 
00442     T f = 0.0;
00443 
00444     for (size_t i = 0; i < n - 1; i++)
00445     {
00446         T a = static_cast<T>(-1.0) * v[i] * std::sin(std::sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0
      )))));
00447         T b = (v[i+1] + static_cast<T>(47)) * std::sin(std::sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (
      v[i]/static_cast<T>(2.0)))));
00448         f += a - b;
00449     }
00450 
00451     return f;
00452 }
00453 
00454 // ================================================
00455 
00463 template <class T>
00464 T mfunc::Functions<T>::rana(T* v, size_t n)
00465 {
00466     fCounterInc(_ranaId);
00467 
00468     T f = 0.0;
00469 
00470     for (size_t i = 0; i < n - 1; i++)
00471     {
00472         T a = v[i] * std::sin(std::sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * std::cos(
      std::sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00473         T b = (v[i+1] + static_cast<T>(1.0)) * std::cos(std::sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1
      .0)))) * std::sin(std::sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
00474         f += a + b;
00475     }
00476 
00477     return f;
00478 }
00479 
00480 // ================================================
00481 
00489 template <class T>
00490 T mfunc::Functions<T>::pathological(T* v, size_t n)
00491 {
00492     fCounterInc(_pathologicalId);
00493 
00494     T f = 0.0;
00495 
00496     for (size_t i = 0; i < n - 1; i++)
00497     {
00498         T a = std::sin(std::sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
00499         a = (a*a) - static_cast<T>(0.5);
00500         T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
00501         b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
00502         f += static_cast<T>(0.5) + (a/b);
00503     }
00504 
00505     return f;
00506 }
00507 
00508 // ================================================
00509 
00517 template <class T>
00518 T mfunc::Functions<T>::michalewicz(T* v, size_t n)
00519 {
00520     fCounterInc(_michalewiczId);
00521 
00522     T f = 0.0;
00523 
00524     for (size_t i = 0; i < n; i++)
00525     {
00526         f += std::sin(v[i]) * std::pow(std::sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)),
      static_cast<T>(20));
00527     }
00528 
00529     return -1.0 * f;
00530 }
00531 
00532 // ================================================
00533 
00541 template <class T>
00542 T mfunc::Functions<T>::mastersCosineWave(T* v, size_t n)
00543 {
00544     fCounterInc(_mastersCosineWaveId);
00545 
00546     T f = 0.0;
00547 
00548     for (size_t i = 0; i < n - 1; i++)
```

```
00549     {
00550         T a = std::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]));
00551         T b = std::cos(static_cast<T>(4) * std::sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v
      [i+1]));
00552         f += a * b;
00553     }
00554 
00555     return static_cast<T>(-1.0) * f;
00556 }
00557 
00558 // ================================================
00559 
00567 template <class T>
00568 T mfunc::Functions<T>::quartic(T* v, size_t n)
00569 {
00570     fCounterInc(_quarticId);
00571 
00572     T f = 0.0;
00573 
00574     for (size_t i = 0; i < n; i++)
00575     {
00576         f += (i+1) * v[i] * v[i] * v[i] * v[i];
00577     }
00578 
00579     return f;
00580 }
00581 
00582 // ================================================
00583 
00587 template <class T>
00588 T  mfunc::Functions<T>::w(T x)
00589 {
00590     return static_cast<T>(1.0) + (x - static_cast<T>(1.0)) / static_cast<T>(4.0);
00591 }
00592 
00600 template <class T>
00601 T mfunc::Functions<T>::levy(T* v, size_t n)
00602 {
00603     fCounterInc(_levyId);
00604 
00605     T f = 0.0;
00606 
00607     for (size_t i = 0; i < n - 1; i++)
00608     {
00609         T a = w(v[i]) - static_cast<T>(1.0);
00610         a *= a;
00611         T b = std::sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
00612         b *= b;
00613         T c = w(v[n - 1]) - static_cast<T>(1.0);
00614         c *= c;
00615         T d = std::sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
00616         d *= d;
00617         f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
00618     }
00619 
00620     T e = std::sin(static_cast<T>(M_PI) * w(v[0]));
00621     return e*e + f;
00622 }
00623 
00624 // ================================================
00625 
00633 template <class T>
00634 T mfunc::Functions<T>::step(T* v, size_t n)
00635 {
00636     fCounterInc(_stepId);
00637 
00638     T f = 0.0;
00639 
00640     for (size_t i = 0; i < n; i++)
00641     {
00642         T a = std::abs(v[i]) + static_cast<T>(0.5);
00643         f += a * a;
00644     }
00645 
00646     return f;
00647 }
00648 
00649 // ================================================
00650 
00658 template <class T>
00659 T mfunc::Functions<T>::alpine(T* v, size_t n)
00660 {
00661     fCounterInc(_alpineId);
00662 
00663     T f = 0.0;
00664 
```

```
00665     for (size_t i = 0; i < n; i++)
00666     {
00667         f += std::abs(v[i] * std::sin(v[i]) + static_cast<T>(0.1)*v[i]);
00668     }
00669
00670     return f;
00671 }
00672
00673 // ================================================
00674
00684 template <class T>
00685 mfunc::mfuncPtr<T> mfunc::Functions<T>::get(unsigned int f)
00686 {
00687     switch (f)
00688     {
00689         case _schwefelId:
00690             return Functions<T>::schwefel;
00691         case _dejongId:
00692             return Functions<T>::dejong;
00693         case _rosenbrokId:
00694             return Functions<T>::rosenbrok;
00695         case _rastriginId:
00696             return Functions<T>::rastrigin;
00697         case _griewangkId:
00698             return Functions<T>::griewangk;
00699         case _sineEnvelopeSineWaveId:
00700             return Functions<T>::sineEnvelopeSineWave;
00701         case _stretchedVSineWaveId:
00702             return Functions<T>::stretchedVSineWave;
00703         case _ackleysOneId:
00704             return Functions<T>::ackleysOne;
00705         case _ackleysTwoId:
00706             return Functions<T>::ackleysTwo;
00707         case _eggHolderId:
00708             return Functions<T>::eggHolder;
00709         case _ranaId:
00710             return Functions<T>::rana;
00711         case _pathologicalId:
00712             return Functions<T>::pathological;
00713         case _michalewiczId:
00714             return Functions<T>::michalewicz;
00715         case _mastersCosineWaveId:
00716             return Functions<T>::mastersCosineWave;
00717         case _quarticId:
00718             return Functions<T>::quartic;
00719         case _levyId:
00720             return Functions<T>::levy;
00721         case _stepId:
00722             return Functions<T>::step;
00723         case _alpineId:
00724             return Functions<T>::alpine;
00725         default:
00726             return nullptr;
00727     }
00728 }
00729
00730 // ================================================
00731
00742 template <class T>
00743 bool mfunc::Functions<T>::exec(unsigned int f, T* v, size_t n, T& outResult)
00744 {
00745     auto fPtr = get(f);
00746     if (fPtr == nullptr) return false;
00747
00748     outResult = fPtr(v, n);
00749     return true;
00750 }
00751
00757 template <class T>
00758 size_t mfunc::Functions<T>::getCallCounter(unsigned int f)
00759 {
00760     if (f == 0 || f > _NUM_FUNCTIONS)
00761         return 0;
00762
00763     return fCallCounters[f - 1];
00764 }
00765
00769 template <class T>
00770 void mfunc::Functions<T>::resetCallCounters()
00771 {
00772     for (size_t i = 0; i < _NUM_FUNCTIONS; i++)
00773         fCallCounters[i] = 0;
00774 }
00775
00779 template <class T>
00780 void mfunc::Functions<T>::fCounterInc(unsigned int f)
00781 {
```

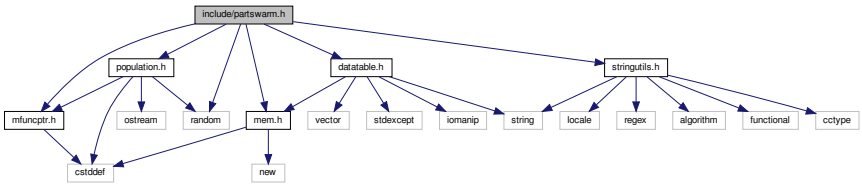
```
00782     if (!fCountersInit)
00783     {
00784         resetCallCounters();
00785         fCountersInit = true;
00786     }
00787     else if (f == 0 || f > _NUM_FUNCTIONS)
00788     {
00789         return;
00790     }
00791 
00792     fCallCounters[f - 1] += 1;
00793 }
00794 
00795 #endif
00796 
00797 // ========================
00798 // End of mfunctions.h
00799 // ========================
```

6.17 include/partswarm.h File Reference

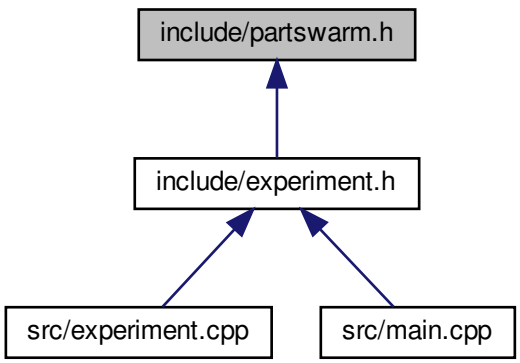
Contains the ParticleSwarm class, which runs the particle swarm algorithm using the given parameters.

```
#include "population.h"
#include "mfuncptr.h"
#include "datatable.h"
#include "random"
#include "mem.h"
#include "stringutils.h"
```

Include dependency graph for partswarm.h:



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



Classes

- struct mfunc::Particle< T >

  The Particle struct is a simple data structure used to store the global best particle along with it's fitness.
- struct mfunc::PSParams< T >

  The PSParams struct contains various parameters that are required to be passed to the ParticleSwarm.run() method.
- class mfunc::ParticleSwarm< T >

  The ParticleSwarm class runs the particle swarm algorithm with the given parameters passed to the run() method.

Namespaces

- mfunc

Macros

- #define POPFILE_GEN_PATTERN "%GEN%"

6.17.1 Detailed Description

Contains the ParticleSwarm class, which runs the particle swarm algorithm using the given parameters.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-05-10

Copyright

Copyright (c) 2019

Definition in file partswarm.h.

6.17.2 Macro Definition Documentation

6.17.2.1 POPFILE_GEN_PATTERN

```
#define POPFILE_GEN_PATTERN "%GEN%"
```

Definition at line 23 of file partswarm.h.

Referenced by mfunc::ParticleSwarm< T >::run().

6.18 partswarm.h

```
00001
00013 #ifndef __PARTSWARM_H
00014 #define __PARTSWARM_H
00015
00016 #include "population.h"
00017 #include "mfuncptr.h"
00018 #include "datatable.h"
00019 #include "random"
00020 #include "mem.h"
00021 #include "stringutils.h"
00022
00023 #define POPFILE_GEN_PATTERN "%GEN%"
00024
00025 namespace mfunc
00026 {
00033     template <class T>
00034     struct Particle
00035     {
00036         T* vector;
00037         T fitness;
00038
00039         Particle()
00040             : vector(nullptr), fitness(0)
00041         {
00042         }
00043     };
00044
00051     template <class T>
00052     struct PSParams
00053     {
00054         std::string popFile; // String file name for population dump file
00055         mdata::DataTable<T>* bestFitnessTable; // Data table for best
      fitness values
00056         mdata::DataTable<T>* worstFitnessTable; // Data table for worst
      fitness values
00057         size_t fitTableCol; // Data table column for best and worst fitness values
00058         mdata::Population<T>* mainPop; // Pointer to main population object
00059         mdata::Population<T>* pbPop; // Pointer to personal best population object
00060         mfuncPtr<T> fPtr; // Function pointer to the objective function being tested
00061         T fMinBound; // Minimum population vector bounds for objective function
00062         T fMaxBound; // Maximum population vector bounds for objective function
00063         unsigned int iterations; // Number of iterations to run search algorithm
00064         double c1; // C1 parameter for particle swarm
00065         double c2; // C2 parameter for particle swarm
00066         double k;  // k dampening factor parameter for particle swarm
00067
00071         PSParams()
00072         {
00073             popFile = "";
00074             bestFitnessTable = nullptr;
00075             worstFitnessTable = nullptr;
00076             fitTableCol = 0;
00077             mainPop = nullptr;
00078             pbPop = nullptr;
00079             fPtr = nullptr;
00080             fMinBound = 0;
00081             fMaxBound = 0;
00082             iterations = 0;
00083             c1 = 0;
00084             c2 = 0;
00085             k = 0;
00086         }
00087     };
00088
00095     template <class T>
00096     class ParticleSwarm
00097     {
00098     public:
```

```
00099         ParticleSwarm();
00100         ~ParticleSwarm() = default;
00101         int run(PSParams<T> params);
00102     private:
00103         std::random_device seed;
00104         std::mt19937 engine;
00105         std::uniform_real_distribution<double> rchance;
00106 
00107         void updateParticle(PSParams<T>& p, const Particle<T>& globalBest, T**
      velMatrix, size_t pIndex);
00108         void randomizeVelocity(T** vMatrix, size_t popSize, size_t dimSize, T fMin, T fMax);
00109     };
00110 }
00111 
00116 template <class T>
00117 mfunc::ParticleSwarm<T>::ParticleSwarm()
00118     : seed(), engine(seed()), rchance(0, 1)
00119 {
00120 }
00121 
00129 template <class T>
00130 int mfunc::ParticleSwarm<T>::run(PSParams<T> p)
00131 {
00132     if (p.mainPop == nullptr || p.pbPop == nullptr || p.fPtr == nullptr)
00133         return 1;
00134 
00135     // Get population information
00136     const size_t popSize = p.mainPop->getPopulationSize();
00137     const size_t dimSize = p.mainPop->getDimensionsSize();
00138 
00139     if (popSize != p.pbPop->getPopulationSize() ||
00140         dimSize != p.pbPop->getDimensionsSize())
00141         return 2;
00142 
00143 
00144     // Construct global best particle and allocate gBest vector
00145     Particle<T> globalBest;
00146     globalBest.vector = util::allocArray<T>(dimSize);
00147 
00148     // Allocate velocity matrix
00149     T** velocityMatrix = util::allocMatrix<T>(popSize, dimSize);
00150 
00151     if (globalBest.vector == nullptr || velocityMatrix == nullptr)
00152         return 3;
00153 
00154     if (!p.mainPop->generate(p.fMinBound, p.fMaxBound))
00155         return 4;
00156 
00157     if (!p.mainPop->calcAllFitness(p.fPtr))
00158         return 5;
00159 
00160     if (!p.pbPop->copyAllFrom(p.mainPop))
00161         return 6;
00162 
00163 
00164     // Randomize the velocities for all particles
00165     randomizeVelocity(velocityMatrix, popSize, dimSize, p.fMinBound, p.
      fMaxBound);
00166 
00167     auto bestFitIndex = p.mainPop->getBestFitnessIndex();
00168     util::copyArray<T>(p.mainPop->getPopulationPtr(bestFitIndex), globalBest.
      vector, dimSize);
00169     globalBest.fitness = p.mainPop->getFitness(bestFitIndex);
00170 
00171     for (unsigned int iter = 0; iter < p.iterations; iter++)
00172     {
00173         for (size_t pIndex = 0; pIndex < popSize; pIndex++)
00174         {
00175             // Update the particles and their velocities
00176             updateParticle(p, globalBest, velocityMatrix, pIndex);
00177         }
00178 
00179         // Get the index of current the best solution, and the associated fitness
00180         bestFitIndex = p.mainPop->getBestFitnessIndex();
00181         T bestFitVal = p.mainPop->getFitness(bestFitIndex);
00182 
00183         // Update global best if current best is better
00184         if (bestFitVal < globalBest.fitness)
00185         {
00186             util::copyArray<T>(p.mainPop->getPopulationPtr(bestFitIndex), globalBest.
      vector, dimSize);
00187             globalBest.fitness = bestFitVal;
00188         }
00189 
00190         // Store best fitness for this iteration
00191         if (p.bestFitnessTable != nullptr)
00192             p.bestFitnessTable->setEntry(iter, p.fitTableCol, globalBest.
```

```
    fitness);
00193
00194         // Store worst fitness for this iteration
00195         if (p.worstFitnessTable != nullptr)
00196             p.worstFitnessTable->setEntry(iter, p.fitTableCol, p.
    mainPop->getWorstFitness());
00197
00198         // Dump population vectors to file
00199         if (!p.popFile.empty())
00200             p.mainPop->outputPopulationCsv(util::s_replace(p.popFile, std::string(
    POPFILE_GEN_PATTERN), std::to_string(iter)));
00201     }
00202
00203     util::releaseArray<T>(globalBest.vector);
00204     util::releaseMatrix<T>(velocityMatrix, popSize);
00205
00206     return 0;
00207 }
00208
00218 template <class T>
00219 void mfunc::ParticleSwarm<T>::updateParticle(
    PSParams<T>& p, const Particle<T>& globalBest, T** velMatrix, size_t pIndex)
00220 {
00221     const size_t dimSize = p.mainPop->getDimensionsSize();
00222     auto pBestVector = p.pbPop->getPopulationPtr(pIndex);
00223     auto curVector = p.mainPop->getPopulationPtr(pIndex);
00224
00225     // Update particle's velocity and position
00226     for (size_t d = 0; d < dimSize; d++)
00227     {
00228         velMatrix[pIndex][d] += p.c1 * rchance(engine) * (pBestVector[d] - curVector[d]) +
00229             p.c2 * rchance(engine) * (globalBest.vector[d] - curVector[d]);
00230         velMatrix[pIndex][d] *= p.k;
00231
00232         curVector[d] += velMatrix[pIndex][d];
00233
00234         if (curVector[d] < p.fMinBound)
00235             curVector[d] = p.fMinBound;
00236         else if (curVector[d] > p.fMaxBound)
00237             curVector[d] = p.fMaxBound;
00238     }
00239
00240     p.mainPop->calcFitness(pIndex, p.fPtr);
00241     T newFitness = p.mainPop->getFitness(pIndex);
00242     T pbFitness = p.pbPop->getFitness(pIndex);
00243
00244     // Update personal best if current position is better
00245     if (newFitness < pbFitness)
00246     {
00247         p.pbPop->copyFrom(p.mainPop, pIndex, pIndex);
00248     }
00249 }
00250
00261 template <class T>
00262 void mfunc::ParticleSwarm<T>::randomizeVelocity(T** vMatrix,
    size_t popSize, size_t dimSize, T fMin, T fMax)
00263 {
00264     std::uniform_real_distribution<T> velDist(0, 0.5 * (fMax - fMin));
00265
00266     for (size_t s = 0; s < popSize; s++)
00267     {
00268         for (size_t d = 0; d < dimSize; d++)
00269         {
00270             vMatrix[s][d] = velDist(engine);
00271         }
00272     }
00273 }
00274
00275
00276
00277 #endif
```

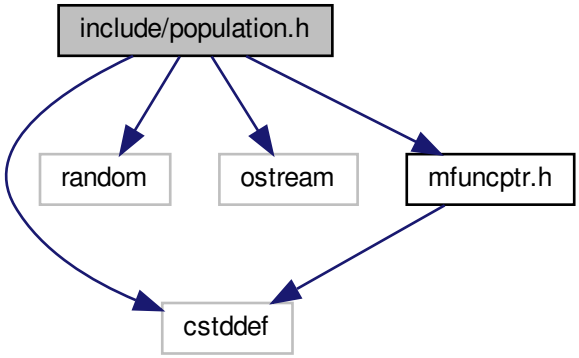
6.19 include/population.h File Reference

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

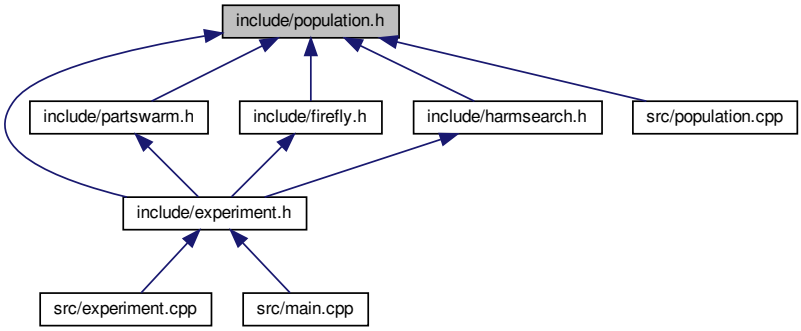
```
#include <cstddef>
#include <random>
#include <ostream>
```

```
#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

6.19.1 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

Definition in file population.h.

6.20 population.h

```
00001
00012 #ifndef __POPULATION_H
00013 #define __POPULATION_H
00014
00015 #include <cstddef> // size_t definition
00016 #include <random>
00017 #include <ostream>
00018 #include "mfuncptr.h"
00019
00020 namespace mdata
00021 {
00028     template<class T>
00029     class Population
00030     {
00031     public:
00032         Population(size_t popSize, size_t dimensions);
00033         ~Population();
00034
00035         bool isReady();
00036         size_t getPopulationSize();
00037         size_t getDimensionsSize();
00038         T* getPopulationPtr(size_t popIndex);
00039         T* getBestPopulationPtr();
00040
00041         bool generate(T minBound, T maxBound);
00042         bool generateSingle(size_t popIndex, T minBound, T maxBound);
00043         bool setFitness(size_t popIndex, T value);
00044         bool calcFitness(size_t popIndex, mfunc::mfuncPtr<T> funcPtr);
00045         bool calcAllFitness(mfunc::mfuncPtr<T> funcPtr);
00046
00047         T getFitness(size_t popIndex);
00048         T* getFitnessPtr(size_t popIndex);
00049
00050         T* getBestFitnessPtr();
00051         size_t getBestFitnessIndex();
00052         T getBestFitness();
00053         size_t getWorstFitnessIndex();
00054         T getWorstFitness();
00055
00056         void sortFitnessAscend();
00057         void sortFitnessDescend();
00058
```

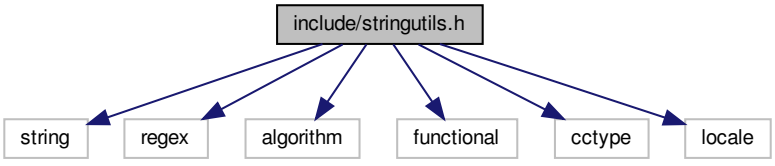
```
00059         bool copyFrom(Population<T>* srcPtr, size_t srcIndex, size_t destIndex);
00060         bool copyAllFrom(Population<T>* srcPtr);
00061         bool copyPopulation(T* src, size_t destIndex);
00062 
00063         void outputPopulation(std::ostream& outStream, const char* delim, const char*
      lineBreak);
00064         void outputFitness(std::ostream& outStream, const char* delim, const char* lineBreak);
00065 
00066         bool outputPopulationCsv(std::string filePath);
00067     private:
00068         const size_t popSize;
00069         const size_t popDim;
00071         T** popMatrix;
00072         T* popFitness;
00074         std::random_device rdev;
00075         std::mt19937 rgen;
00077         bool allocPopMatrix();
00078         void releasePopMatrix();
00079 
00080         bool allocPopFitness();
00081         void releasePopFitness();
00082 
00083         void qs_swapval(T& a, T& b);
00084         void qs_swapptr(T*& a, T*& b);
00085         long part_fit_ascend(long low, long high);
00086         void qs_fit_ascend(long low, long high);
00087 
00088         long part_fit_descend(long low, long high);
00089         void qs_fit_descend(long low, long high);
00090     };
00091 }
00092 
00093 #endif
00094 
00095 // ========================
00096 // End of population.h
00097 // ========================
```

6.21 include/stringutils.h File Reference

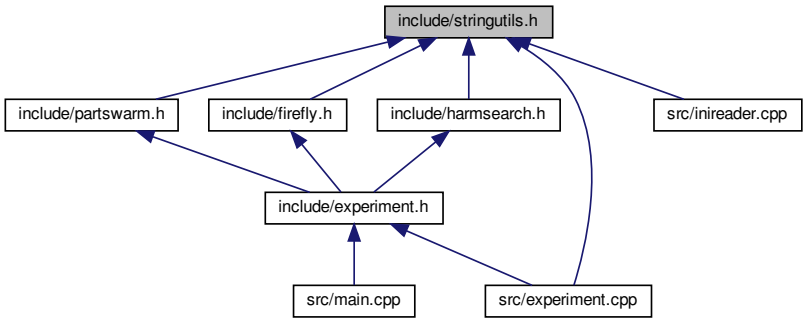
Contains various string manipulation helper functions.

```
#include <string>
#include <regex>
#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

6.21.1 Detailed Description

Contains various string manipulation helper functions.

Author

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

Date

2019-04-01

Definition in file stringutils.h.

6.22 stringutils.h

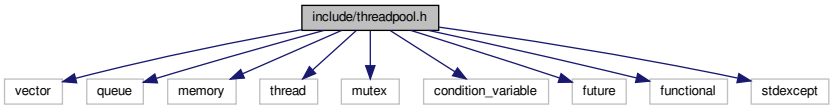
```
00001
00008 #ifndef __STRINGUTILS_H
00009 #define __STRINGUTILS_H
00010
00011 #include <string>
00012 #include <regex>
00013 #include <algorithm>
00014 #include <functional>
00015 #include <cctype>
00016 #include <locale>
00017
00018 namespace util
00019 {
00028     static inline std::string s_replace(std::string input, std::string pattern, std::string replacement)
00029     {
00030         pattern = std::string("\\") + pattern;
00031         return std::regex_replace(input, std::regex(pattern), replacement);
00032     }
00033
```

```
00034     // ====================================================
00035     // The string functions below were written by Evan Teran
00036     // from Stack Overflow:
00037     // https://stackoverflow.com/questions/216823/whats-the-best-way-to-trim-stdstring
00038     // ====================================================
00039
00040     // trim from start (in place)
00041     static inline void s_ltrim(std::string &s) {
00042         s.erase(s.begin(), std::find_if(s.begin(), s.end(),
00043                 std::not1(std::ptr_fun<int, int>(std::isspace))));
00044     }
00045
00046     // trim from end (in place)
00047     static inline void s_rtrim(std::string &s) {
00048         s.erase(std::find_if(s.rbegin(), s.rend(),
00049                 std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
00050     }
00051
00052     // trim from both ends (in place)
00053     static inline void s_trim(std::string &s) {
00054         s_ltrim(s);
00055         s_rtrim(s);
00056     }
00057
00058     // trim from start (copying)
00059     static inline std::string s_ltrim_copy(std::string s) {
00060         s_ltrim(s);
00061         return s;
00062     }
00063
00064     // trim from end (copying)
00065     static inline std::string s_rtrim_copy(std::string s) {
00066         s_rtrim(s);
00067         return s;
00068     }
00069
00070     // trim from both ends (copying)
00071     static inline std::string s_trim_copy(std::string s) {
00072         s_trim(s);
00073         return s;
00074     }
00075 }
00076 #endif
00077
00078 // ========================
00079 // End of stringutils.h
00080 // ========================
```

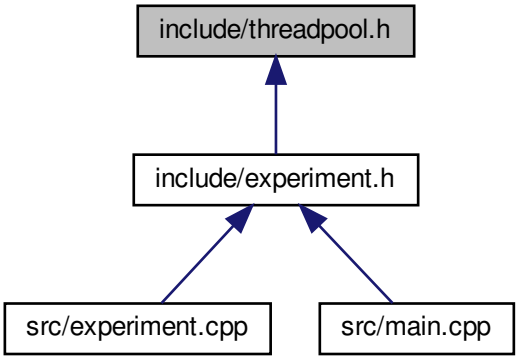
6.23 include/threadpool.h File Reference

```
#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

6.24 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 // ==========================
```

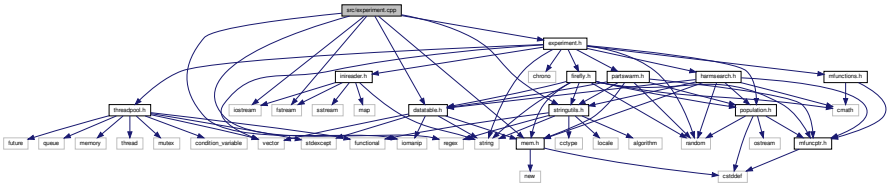
6.25 src/experiment.cpp File Reference

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

```
#include <iostream>
#include <fstream>
```

```
#include <iomanip>
#include <regex>
#include "experiment.h"
#include "datatable.h"
#include "stringutils.h"
#include "mem.h"
```

Include dependency graph for experiment.cpp:



Macros

- #define INI_TEST_SECTION "test"
- #define INI_PSO_SECTION "particle_swarm"
- #define INI_FA_SECTION "firefly"
- #define INI_HS_SECTION "harmony_search"
- #define INI_FUNC_RANGE_SECTION "function_range"
- #define INI_TEST_POPULATION "population"
- #define INI_TEST_DIMENSIONS "dimensions"
- #define INI_TEST_ITERATIONS "iterations"
- #define INI_TEST_NUMTHREADS "num_threads"
- #define INI_TEST_ALGORITHM "algorithm"
- #define INI_TEST_RESULTSFILE "results_file"
- #define INI_TEST_WORSTFITNESSFILE "worst_fit_file"
- #define INI_TEST_EXECTIMESFILE "exec_times_file"
- #define INI_TEST_FUNCCALLSFILE "func_calls_file"
- #define INI_TEST_POPULATIONFILE "population_file"
- #define INI_PSO_C1 "c1"
- #define INI_PSO_C2 "c2"
- #define INI_PSO_K "k"
- #define INI_FA_ALPHA "alpha"
- #define INI_FA_BETAMIN "betamin"
- #define INI_FA_GAMMA "gamma"
- #define INI_HS_HMCR "hmcr"
- #define INI_HS_PAR "par"
- #define INI_HS_BW "bw"
- #define PARAM_DEFAULT_PSO_C1 0.8
- #define PARAM_DEFAULT_PSO_C2 1.2
- #define PARAM_DEFAULT_PSO_K 1.0
- #define PARAM_DEFAULT_FA_ALPHA 0.5
- #define PARAM_DEFAULT_FA_BETAMIN 0.2
- #define PARAM_DEFAULT_FA_GAMMA 0.1
- #define PARAM_DEFAULT_HS_HMCR 0.9
- #define PARAM_DEFAULT_HS_PAR 0.4
- #define PARAM_DEFAULT_HS_BW 0.2
- #define RESULTSFILE_ALG_PATTERN "%ALG%"

6.25.1 Detailed Description

Implementation 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

0.4

Date

2019-04-01

Copyright

Definition in file experiment.cpp.

6.25.2 Macro Definition Documentation

6.25.2.1 INI_FA_ALPHA

```
#define INI_FA_ALPHA "alpha"
```

Definition at line 44 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.2 INI_FA_BETAMIN

```
#define INI_FA_BETAMIN "betamin"
```

Definition at line 45 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.3 INI_FA_GAMMA

```
#define INI_FA_GAMMA "gamma"
```

Definition at line 46 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.4 INI_FA_SECTION

```
#define INI_FA_SECTION "firefly"
```

Definition at line 25 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.5 INI_FUNC_RANGE_SECTION

```
#define INI_FUNC_RANGE_SECTION "function_range"
```

Definition at line 27 of file experiment.cpp.

6.25.2.6 INI_HS_BW

```
#define INI_HS_BW "bw"
```

Definition at line 50 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.7 INI_HS_HMCR

```
#define INI_HS_HMCR "hmcr"
```

Definition at line 48 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.8 INI_HS_PAR

```
#define INI_HS_PAR "par"
```

Definition at line 49 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.9 INI_HS_SECTION

```
#define INI_HS_SECTION "harmony_search"
```

Definition at line 26 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.10 INI_PSO_C1

```
#define INI_PSO_C1 "c1"
```

Definition at line 40 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.11 INI_PSO_C2

```
#define INI_PSO_C2 "c2"
```

Definition at line 41 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.12 INI_PSO_K

```
#define INI_PSO_K "k"
```

Definition at line 42 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.13 INI_PSO_SECTION

```
#define INI_PSO_SECTION "particle_swarm"
```

Definition at line 24 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.14 INI_TEST_ALGORITHM

```
#define INI_TEST_ALGORITHM "algorithm"
```

Definition at line 33 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.15 INI_TEST_DIMENSIONS

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

Definition at line 30 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.16 INI_TEST_EXECTIMESFILE

```
#define INI_TEST_EXECTIMESFILE "exec_times_file"
```

Definition at line 36 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.17 INI_TEST_FUNCCALLSFILE

```
#define INI_TEST_FUNCCALLSFILE "func_calls_file"
```

Definition at line 37 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.18 INI_TEST_ITERATIONS

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

Definition at line 31 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.19 INI_TEST_NUMTHREADS

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

Definition at line 32 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.20 INI_TEST_POPULATION

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

Definition at line 29 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.21 INI_TEST_POPULATIONFILE

```
#define INI_TEST_POPULATIONFILE "population_file"
```

Definition at line 38 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.22 INI_TEST_RESULTSFILE

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

Definition at line 34 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.23 INI_TEST_SECTION

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

Definition at line 23 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.24 INI_TEST_WORSTFITNESSFILE

```
#define INI_TEST_WORSTFITNESSFILE "worst_fit_file"
```

Definition at line 35 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::init().

6.25.2.25 PARAM_DEFAULT_FA_ALPHA

```
#define PARAM_DEFAULT_FA_ALPHA 0.5
```

Definition at line 57 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.26 PARAM_DEFAULT_FA_BETAMIN

```
#define PARAM_DEFAULT_FA_BETAMIN 0.2
```

Definition at line 58 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.27 PARAM_DEFAULT_FA_GAMMA

```
#define PARAM_DEFAULT_FA_GAMMA 0.1
```

Definition at line 59 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.28 PARAM_DEFAULT_HS_BW

```
#define PARAM_DEFAULT_HS_BW 0.2
```

Definition at line 63 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.29 PARAM_DEFAULT_HS_HMCR

```
#define PARAM_DEFAULT_HS_HMCR 0.9
```

Definition at line 61 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.30 PARAM_DEFAULT_HS_PAR

```
#define PARAM_DEFAULT_HS_PAR 0.4
```

Definition at line 62 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.31 PARAM_DEFAULT_PSO_C1

```
#define PARAM_DEFAULT_PSO_C1 0.8
```

Definition at line 53 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.32 PARAM_DEFAULT_PSO_C2

```
#define PARAM_DEFAULT_PSO_C2 1.2
```

Definition at line 54 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.33 PARAM_DEFAULT_PSO_K

```
#define PARAM_DEFAULT_PSO_K 1.0
```

Definition at line 55 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testHS().

6.25.2.34 RESULTSFILE_ALG_PATTERN

```
#define RESULTSFILE_ALG_PATTERN "%ALG%"
```

Definition at line 65 of file experiment.cpp.

Referenced by mfunc::Experiment< T >::testFA(), mfunc::Experiment< T >::testHS(), and mfunc::Experiment< T >::testPS().

6.26 experiment.cpp

```
00001
00013 #include <iostream>
00014 #include <fstream>
00015 #include <iomanip>
00016 #include <regex>
00017 #include "experiment.h"
00018 #include "datatable.h"
00019 #include "stringutils.h"
00020 #include "mem.h"
00021
00022 // Ini file string sections and keys
00023 #define INI_TEST_SECTION "test"
00024 #define INI_PSO_SECTION "particle_swarm"
00025 #define INI_FA_SECTION "firefly"
00026 #define INI_HS_SECTION "harmony_search"
00027 #define INI_FUNC_RANGE_SECTION "function_range"
00028
00029 #define INI_TEST_POPULATION "population"
00030 #define INI_TEST_DIMENSIONS "dimensions"
00031 #define INI_TEST_ITERATIONS "iterations"
00032 #define INI_TEST_NUMTHREADS "num_threads"
00033 #define INI_TEST_ALGORITHM "algorithm"
00034 #define INI_TEST_RESULTSFILE "results_file"
00035 #define INI_TEST_WORSTFITNESSFILE "worst_fit_file"
00036 #define INI_TEST_EXECTIMESFILE "exec_times_file"
00037 #define INI_TEST_FUNCCALLSFILE "func_calls_file"
00038 #define INI_TEST_POPULATIONFILE "population_file"
00039
00040 #define INI_PSO_C1 "c1"
00041 #define INI_PSO_C2 "c2"
00042 #define INI_PSO_K "k"
00043
00044 #define INI_FA_ALPHA "alpha"
00045 #define INI_FA_BETAMIN "betamin"
00046 #define INI_FA_GAMMA "gamma"
00047
00048 #define INI_HS_HMCR "hmcr"
00049 #define INI_HS_PAR "par"
00050 #define INI_HS_BW "bw"
00051
00052 // Default algorithm parameters
00053 #define PARAM_DEFAULT_PSO_C1 0.8
00054 #define PARAM_DEFAULT_PSO_C2 1.2
00055 #define PARAM_DEFAULT_PSO_K 1.0
00056
00057 #define PARAM_DEFAULT_FA_ALPHA 0.5
00058 #define PARAM_DEFAULT_FA_BETAMIN 0.2
00059 #define PARAM_DEFAULT_FA_GAMMA 0.1
00060
```

```
00061 #define PARAM_DEFAULT_HS_HMCR 0.9
00062 #define PARAM_DEFAULT_HS_PAR 0.4
00063 #define PARAM_DEFAULT_HS_BW 0.2
00064
00065 #define RESULTSFILE_ALG_PATTERN "%ALG%"
00066
00067 using namespace std;
00068 using namespace std::chrono;
00069 using namespace mfunc;
00070
00074 template<class T>
00075 Experiment<T>::Experiment()
00076     : vBounds(nullptr), tPool(nullptr), resultsFile(""), execTimesFile(""), iterations(0)
00077 {
00078 }
00079
00084 template<class T>
00085 Experiment<T>::~Experiment()
00086 {
00087     releaseThreadPool();
00088     releasePopulationPool();
00089     releaseVBounds();
00090 }
00091
00100 template<class T>
00101 bool Experiment<T>::init(const char* paramFile)
00102 {
00103     try
00104     {
00105         // Open and parse parameters file
00106         if (!iniParams.openFile(paramFile))
00107         {
00108             cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
00109             return false;
00110         }
00111
00112         // Extract test parameters from ini file
00113         long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_POPULATION);
00114         long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_DIMENSIONS);
00115         long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION,
      INI_TEST_ITERATIONS);
00116         long numberThreads = iniParams.getEntryAs<long>(
      INI_TEST_SECTION, INI_TEST_NUMTHREADS);
00117         unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(
      INI_TEST_SECTION, INI_TEST_ALGORITHM);
00118         resultsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_RESULTSFILE);
00119         worstFitnessFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_WORSTFITNESSFILE);
00120         execTimesFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_EXECTIMESFILE);
00121         funcCallsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_FUNCCALLSFILE);
00122         populationsFile = iniParams.getEntry(INI_TEST_SECTION,
      INI_TEST_POPULATIONFILE);
00123
00124         // Verify test parameters
00125         if (numberSol <= 0)
00126         {
00127             cerr << "Experiment init failed: Param file [test]->"
00128                 << INI_TEST_POPULATION << " entry missing or out of bounds: " <<
      paramFile << endl;
00129             return false;
00130         }
00131         else if (numberDim <= 0)
00132         {
00133             cerr << "Experiment init failed: Param file [test]->"
00134                 << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " <<
      paramFile << endl;
00135             return false;
00136         }
00137         else if (numberIter <= 0)
00138         {
00139             cerr << "Experiment init failed: Param file [test]->"
00140                 << INI_TEST_ITERATIONS << " entry missing or out of bounds: " <<
      paramFile << endl;
00141             return false;
00142         }
00143         else if (numberThreads <= 0)
00144         {
00145             cerr << "Experiment init failed: Param file [test]->"
00146                 << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " <<
      paramFile << endl;
00147             return false;
00148         }
```

```
00149         else if (selectedAlg >= static_cast<unsigned int>(Algorithm::Count))
00150         {
00151             cerr << "Experiment init failed: Param file [test]->"
00152                 << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile
    << endl;
00153             return false;
00154         }
00155 
00156         // Cast iterations and test algorithm to correct types
00157         iterations = (size_t)numberIter;
00158         selAlg = static_cast<Algorithm>(selectedAlg);
00159 
00160         // Print test parameters to console
00161         cout << "Population size: " << numberSol << endl;
00162         cout << "Dimensions: " << numberDim << endl;
00163         cout << "Iterations: " << iterations << endl;
00164 
00165         // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
00166         if (!allocatePopulationPool((size_t)numberThreads * 2, (size_t)numberSol, (size_t)numberDim))
00167         {
00168             cerr << "Experiment init failed: Unable to allocate populations." << endl;
00169             return false;
00170         }
00171 
00172         // Allocate memory for function vector bounds
00173         if (!allocateVBounds())
00174         {
00175             cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
00176             return false;
00177         }
00178 
00179         // Fill function bounds array with data parsed from iniParams
00180         if (!parseFuncBounds())
00181         {
00182             cerr << "Experiment init failed: Unable to parse vector bounds array." << endl;
00183             return false;
00184         }
00185 
00186         // Allocate thread pool
00187         if (!allocateThreadPool((size_t)numberThreads))
00188         {
00189             cerr << "Experiment init failed: Unable to allocate thread pool." << endl;
00190             return false;
00191         }
00192 
00193         cout << "Started " << numberThreads << " worker threads ..." << endl;
00194 
00195         // Ready to run an experiment
00196         return true;
00197     }
00198     catch (const std::exception& ex)
00199     {
00200         cerr << "Exception occurred while initializing experiment: " << ex.what() << endl;
00201         return false;
00202     }
00203     catch (...)
00204     {
00205         cerr << "Unknown Exception occurred while initializing experiment." << endl;
00206         return false;
00207     }
00208 }
00209 
00216 template<class T>
00217 int Experiment<T>::testAllFunc()
00218 {
00219     // Run the selected algorithm
00220     switch (selAlg)
00221     {
00222     case Algorithm::ParticleSwarm:
00223         return testPS();
00224         break;
00225     case Algorithm::Firefly:
00226         return testFA();
00227         break;
00228     case Algorithm::HarmonySearch:
00229         return testHS();
00230         break;
00231     default:
00232         cout << "Error: Invalid algorithm selected." << endl;
00233         break;
00234     }
00235 
00236     return 1;
00237 }
00238 
00245 template<class T>
00246 int Experiment<T>::testPS()
```

```
00247 {
00248     // Prepare alg parameter template struct and results tables
00249     const PSParams<T> paramTemplate = createPSParamsTemplate();
00250     mdata::DataTable<T> resultsTable(iterations, 18);
00251     mdata::DataTable<T> worstTable(iterations, 18);
00252     mdata::DataTable<T> execTimesTable(1, 18);
00253     mdata::DataTable<T> funcCallsTable(1, 18);
00254     std::vector<std::future<int>> testFutures;
00255 
00256     // Reset objective function call counters
00257     mfunc::Functions<T>::resetCallCounters();
00258 
00259     // Queue up a threaded task for each of the 18 objective functions
00260     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00261     {
00262         // Set results table column labels
00263         auto desc = mfunc::FunctionDesc::get(f);
00264         resultsTable.setColLabel(f - 1, desc);
00265         worstTable.setColLabel(f - 1, desc);
00266         execTimesTable.setColLabel(f - 1, desc);
00267         funcCallsTable.setColLabel(f - 1, desc);
00268 
00269         // Create new parameters struct for current function and set parameters
00270         PSParams<T> params(paramTemplate);
00271         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00272         params.bestFitnessTable = &resultsTable;
00273         params.worstFitnessTable = &worstTable;
00274         params.fitTableCol = f - 1;
00275         params.mainPop = nullptr;
00276         params.pbPop = nullptr;
00277         params.fPtr = mfunc::Functions<T>::get(f);
00278         params.fMinBound = vBounds[f-1].min;
00279         params.fMaxBound = vBounds[f-1].max;
00280         params.iterations = iterations;
00281 
00282         // Add search algorithm run to thread pool queue
00283         testFutures.emplace_back(
00284                 tPool->enqueue(&Experiment<T>::runPSThreaded, this,
      params, &execTimesTable, 0, f - 1)
00285         );
00286     }
00287 
00288     cout << "Executing particle swarm ..." << endl << flush;
00289 
00290     // Wait for threads to finish running all functions
00291     waitThreadFutures(testFutures);
00292     testFutures.clear();
00293 
00294     cout << endl;
00295 
00296     // Output objective function call counter values to .csv file
00297     if (!funcCallsFile.empty())
00298     {
00299         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00300             funcCallsTable.setEntry(0, f - 1,
      mfunc::Functions<T>::getCallCounter(f));
00301 
00302         std::string outFile = util::s_replace(funcCallsFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00303         if (funcCallsTable.exportCSV(outFile.c_str()))
00304             cout << "Function call counts written to: " << outFile << endl;
00305         else
00306             cout << "Unable to function call counts file: " << outFile << endl;
00307     }
00308 
00309     // Output best fitness values to .csv file
00310     if (!resultsFile.empty())
00311     {
00312         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00313         if (resultsTable.exportCSV(outFile.c_str()))
00314             cout << "Best fitness results written to: " << outFile << endl;
00315         else
00316             cout << "Unable to open results file: " << outFile << endl;
00317     }
00318 
00319     // Output worst fitness values to .csv file
00320     if (!worstFitnessFile.empty())
00321     {
00322         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00323         if (worstTable.exportCSV(outFile.c_str()))
00324             cout << "Worst fitness results written to: " << outFile << endl;
00325         else
00326             cout << "Unable to open worst fitness file: " << outFile << endl;
00327     }
00328 
```

```
00329     // Output execution times to .csv file
00330     if (!execTimesFile.empty())
00331     {
00332         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "PSO");
00333         if (execTimesTable.exportCSV(outFile.c_str()))
00334             cout << "Execution times written to: " << outFile << endl;
00335         else
00336             cout << "Unable to open execution times file: " << outFile << endl;
00337     }
00338
00339     return 0;
00340 }
00341
00352 template<class T>
00353 int Experiment<T>::runPSThreaded(PSParams<T> params,
      mdata::DataTable<T>* timesTable, size_t tRow, size_t tCol)
00354 {
00355     // Retrieve population objects from population pool
00356     auto mainPop = popPoolRemove();
00357     auto pbPop = popPoolRemove();
00358     params.mainPop = mainPop;
00359     params.pbPop = pbPop;
00360
00361     high_resolution_clock::time_point t_start = high_resolution_clock::now();
00362
00363     // Run search algorithm with given parameters
00364     ParticleSwarm<T> pswarm;
00365     int ret = pswarm.run(params);
00366
00367     high_resolution_clock::time_point t_end = high_resolution_clock::now();
00368     double execTimeMs = static_cast<double>(duration_cast<nanoseconds>(t_end - t_start).count()) / 1000000.
      0;
00369
00370     // Record execution time
00371     if (timesTable != nullptr)
00372         timesTable->setEntry(tRow, tCol, execTimeMs);
00373
00374     // Place population objects back into the pool to be used by another thread
00375     popPoolAdd(mainPop);
00376     popPoolAdd(pbPop);
00377     return ret;
00378 }
00379
00386 template<class T>
00387 int Experiment<T>::testFA()
00388 {
00389     // Prepare alg parameter template struct and results tables
00390     const FAParams<T> paramTemplate = createFAParamsTemplate();
00391     mdata::DataTable<T> resultsTable(iterations, 18);
00392     mdata::DataTable<T> worstTable(iterations, 18);
00393     mdata::DataTable<T> execTimesTable(1, 18);
00394     mdata::DataTable<T> funcCallsTable(1, 18);
00395     std::vector<std::future<int>> testFutures;
00396
00397     // Reset objective function call counters
00398     mfunc::Functions<T>::resetCallCounters();
00399
00400     // Queue up a threaded task for each of the 18 objective functions
00401     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00402     {
00403         // Set results table column labels
00404         auto desc = mfunc::FunctionDesc::get(f);
00405         resultsTable.setColLabel(f - 1, desc);
00406         worstTable.setColLabel(f - 1, desc);
00407         execTimesTable.setColLabel(f - 1, desc);
00408         funcCallsTable.setColLabel(f - 1, desc);
00409
00410         // Create new parameters struct for current function and set parameters
00411         FAParams<T> params(paramTemplate);
00412         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00413         params.bestFitnessTable = &resultsTable;
00414         params.worstFitnessTable = &worstTable;
00415         params.fitTableCol = f - 1;
00416         params.mainPop = nullptr;
00417         params.fPtr = mfunc::Functions<T>::get(f);
00418         params.fMinBound = vBounds[f-1].min;
00419         params.fMaxBound = vBounds[f-1].max;
00420         params.iterations = iterations;
00421
00422         // Add search algorithm run to thread pool queue
00423         testFutures.emplace_back(
00424                 tPool->enqueue(&Experiment<T>::runFAThreaded, this,
      params, &execTimesTable, 0, f - 1)
00425         );
00426     }
00427
```

```
00428     cout << "Executing firefly ..." << endl << flush;
00429
00430     // Wait for all threads to finish
00431     waitThreadFutures(testFutures);
00432     testFutures.clear();
00433
00434     cout << endl;
00435
00436     // Output objective function call counter values to .csv file
00437     if (!funcCallsFile.empty())
00438     {
00439         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00440             funcCallsTable.setEntry(0, f - 1,
      mfunc::Functions<T>::getCallCounter(f));
00441
00442         std::string outFile = util::s_replace(funcCallsFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00443         if (funcCallsTable.exportCSV(outFile.c_str()))
00444             cout << "Function call counts written to: " << outFile << endl;
00445         else
00446             cout << "Unable to function call counts file: " << outFile << endl;
00447     }
00448
00449     // Output best fitness values to .csv file
00450     if (!resultsFile.empty())
00451     {
00452         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00453         if (resultsTable.exportCSV(outFile.c_str()))
00454             cout << "Best fitness results written to: " << outFile << endl;
00455         else
00456             cout << "Unable to open results file: " << outFile << endl;
00457     }
00458
00459     // Output worst fitness values to .csv file
00460     if (!worstFitnessFile.empty())
00461     {
00462         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00463         if (worstTable.exportCSV(outFile.c_str()))
00464             cout << "Worst fitness results written to: " << outFile << endl;
00465         else
00466             cout << "Unable to open worst fitness file: " << outFile << endl;
00467     }
00468
00469     // Output execution times to .csv file
00470     if (!execTimesFile.empty())
00471     {
00472         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "FA");
00473         if (execTimesTable.exportCSV(outFile.c_str()))
00474             cout << "Execution times written to: " << outFile << endl;
00475         else
00476             cout << "Unable to open execution times file: " << outFile << endl;
00477     }
00478
00479     return 0;
00480 }
00481
00492 template<class T>
00493 int Experiment<T>::runFAThreaded(FAParams<T> params,
      mdata::DataTable<T>* timesTable, size_t tRow, size_t tCol)
00494 {
00495     // Retrieve population objects from population pool
00496     auto mainPop = popPoolRemove();
00497     auto nextPop = popPoolRemove();
00498     params.mainPop = mainPop;
00499     params.nextPop = nextPop;
00500
00501     high_resolution_clock::time_point t_start = high_resolution_clock::now();
00502
00503     // Run search algorithm with given parameters
00504     Firefly<T> ffly;
00505     int ret = ffly.run(params);
00506
00507     high_resolution_clock::time_point t_end = high_resolution_clock::now();
00508     double execTimeMs = static_cast<double>(duration_cast<nanoseconds>(t_end - t_start).count()) / 1000000.
      0;
00509
00510     // Record execution time
00511     if (timesTable != nullptr)
00512         timesTable->setEntry(tRow, tCol, execTimeMs);
00513
00514     // Place population objects back into the pool to be used by another thread
00515     popPoolAdd(mainPop);
00516     popPoolAdd(nextPop);
00517     return ret;
```

```
00518 }
00519
00526 template<class T>
00527 int Experiment<T>::testHS()
00528 {
00529     // Prepare alg parameter template struct and results tables
00530     const HSParams<T> paramTemplate = createHSParamsTemplate();
00531     mdata::DataTable<T> resultsTable(iterations, 18);
00532     mdata::DataTable<T> worstTable(iterations, 18);
00533     mdata::DataTable<T> execTimesTable(1, 18);
00534     mdata::DataTable<T> funcCallsTable(1, 18);
00535     std::vector<std::future<int>> testFutures;
00536
00537     // Reset objective function call counters
00538     mfunc::Functions<T>::resetCallCounters();
00539
00540     // Queue up a threaded task for each of the 18 objective functions
00541     for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00542     {
00543         // Set results table column labels
00544         auto desc = mfunc::FunctionDesc::get(f);
00545         resultsTable.setColLabel(f - 1, desc);
00546         worstTable.setColLabel(f - 1, desc);
00547         execTimesTable.setColLabel(f - 1, desc);
00548         funcCallsTable.setColLabel(f - 1, desc);
00549
00550         // Create new parameters struct for current function and set parameters
00551         HSParams<T> params(paramTemplate);
00552         params.popFile = util::s_replace(populationsFile, "%FUNC%", std::to_string(f));
00553         params.bestFitnessTable = &resultsTable;
00554         params.worstFitnessTable = &worstTable;
00555         params.fitTableCol = f - 1;
00556         params.mainPop = nullptr;
00557         params.fPtr = mfunc::Functions<T>::get(f);
00558         params.fMinBound = vBounds[f-1].min;
00559         params.fMaxBound = vBounds[f-1].max;
00560         params.iterations = iterations;
00561
00562         // Add search algorithm run to thread pool queue
00563         testFutures.emplace_back(
00564                 tPool->enqueue(&Experiment<T>::runHSThreaded, this,
      params, &execTimesTable, 0, f - 1)
00565         );
00566     }
00567
00568     cout << "Executing harmony search ..." << endl << flush;
00569
00570     waitThreadFutures(testFutures);
00571
00572     // Clear thread futures
00573     testFutures.clear();
00574
00575     cout << endl;
00576
00577     // Output objective function call counter values to .csv file
00578     if (!funcCallsFile.empty())
00579     {
00580         for (unsigned int f = 1; f <= mfunc::NUM_FUNCTIONS; f++)
00581             funcCallsTable.setEntry(0, f - 1,
      mfunc::Functions<T>::getCallCounter(f));
00582
00583         std::string outFile = util::s_replace(funcCallsFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00584         if (funcCallsTable.exportCSV(outFile.c_str()))
00585             cout << "Function call counts written to: " << outFile << endl;
00586         else
00587             cout << "Unable to function call counts file: " << outFile << endl;
00588     }
00589
00590     // Output best fitness values to .csv file
00591     if (!resultsFile.empty())
00592     {
00593         std::string outFile = util::s_replace(resultsFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00594         if (resultsTable.exportCSV(outFile.c_str()))
00595             cout << "Best fitness results written to: " << outFile << endl;
00596         else
00597             cout << "Unable to open results file: " << outFile << endl;
00598     }
00599
00600     // Output worst fitness values to .csv file
00601     if (!worstFitnessFile.empty())
00602     {
00603         std::string outFile = util::s_replace(worstFitnessFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00604         if (worstTable.exportCSV(outFile.c_str()))
00605             cout << "Worst fitness results written to: " << outFile << endl;
```

```
00606         else
00607             cout << "Unable to open worst fitness file: " << outFile << endl;
00608     }
00609
00610     // Output execution times to .csv file
00611     if (!execTimesFile.empty())
00612     {
00613         std::string outFile = util::s_replace(execTimesFile,
      RESULTSFILE_ALG_PATTERN, "HS");
00614         if (execTimesTable.exportCSV(outFile.c_str()))
00615             cout << "Execution times written to: " << outFile << endl;
00616         else
00617             cout << "Unable to open execution times file: " << outFile << endl;
00618     }
00619
00620     return 0;
00621 }
00622
00633 template<class T>
00634 int Experiment<T>::runHSThreaded(HSParams<T> params,
      mdata::DataTable<T>* timesTable, size_t tRow, size_t tCol)
00635 {
00636     // Retrieve population object from population pool
00637     auto mainPop = popPoolRemove();
00638     params.mainPop = mainPop;
00639
00640     high_resolution_clock::time_point t_start = high_resolution_clock::now();
00641
00642     // Run search algorithm with given parameters
00643     HarmonySearch<T> hsearch;
00644     int ret = hsearch.run(params);
00645
00646     high_resolution_clock::time_point t_end = high_resolution_clock::now();
00647     double execTimeMs = static_cast<double>(duration_cast<nanoseconds>(t_end - t_start).count()) / 1000000.
      0;
00648
00649     // Record execution time
00650     if (timesTable != nullptr)
00651         timesTable->setEntry(tRow, tCol, execTimeMs);
00652
00653     // Place population object back into the pool to be used by another thread
00654     popPoolAdd(mainPop);
00655     return ret;
00656 }
00657
00665 template<class T>
00666 int Experiment<T>::waitThreadFutures(std::vector<std::future<int>>&
      testFutures)
00667 {
00668     cout << "Waiting for threads to finish ..." << endl << flush;
00669
00670     const size_t totalFutures = testFutures.size();
00671
00672     // Join all thread futures and get result
00673     for (size_t futIndex = 0; futIndex < testFutures.size(); futIndex++)
00674     {
00675         auto& curFut = testFutures[futIndex];
00676
00677         if (!curFut.valid())
00678         {
00679             // An error occured with one of the threads
00680             cerr << "Error: Thread future invalid.";
00681             tPool->stopAndJoinAll();
00682             return 1;
00683         }
00684
00685         int errCode = curFut.get();
00686         if (errCode)
00687         {
00688             // An error occurred while running the task.
00689             // Bail out of function
00690             cerr << "Error: Threaded function returned error code: " << errCode << endl;
00691             tPool->stopAndJoinAll();
00692             return errCode;
00693         }
00694
00695         cout << futIndex + 1 << ".." << flush;
00696     }
00697
00698     return 0;
00699 }
00700
00707 template<class T>
00708 const PSParams<T> Experiment<T>::createPSParamsTemplate()
00709 {
00710     PSParams<T> retParams;
00711
```

```
00712     retParams.c1 = iniParams.getEntryAs<double>(INI_PSO_SECTION,
      INI_PSO_C1, PARAM_DEFAULT_PSO_C1);
00713     retParams.c2 = iniParams.getEntryAs<double>(INI_PSO_SECTION,
      INI_PSO_C2, PARAM_DEFAULT_PSO_C2);
00714     retParams.k = iniParams.getEntryAs<double>(INI_PSO_SECTION,
      INI_PSO_K, PARAM_DEFAULT_PSO_K);
00715
00716     return retParams;
00717 }
00718
00725 template<class T>
00726 const FAParams<T> Experiment<T>::createFAParamsTemplate()
00727 {
00728     FAParams<T> retParams;
00729
00730     retParams.alpha = iniParams.getEntryAs<double>(INI_FA_SECTION,
      INI_FA_ALPHA, PARAM_DEFAULT_FA_ALPHA);
00731     retParams.betamin = iniParams.getEntryAs<double>(
      INI_FA_SECTION, INI_FA_BETAMIN,
      PARAM_DEFAULT_FA_BETAMIN);
00732     retParams.gamma = iniParams.getEntryAs<double>(INI_FA_SECTION,
      INI_FA_GAMMA, PARAM_DEFAULT_FA_GAMMA);
00733
00734     return retParams;
00735 }
00736
00743 template<class T>
00744 const HSParams<T> Experiment<T>::createHSParamsTemplate()
00745 {
00746     HSParams<T> retParams;
00747
00748     retParams.hmcr = iniParams.getEntryAs<double>(INI_HS_SECTION,
      INI_HS_HMCR, PARAM_DEFAULT_HS_HMCR);
00749     retParams.par = iniParams.getEntryAs<double>(INI_HS_SECTION,
      INI_HS_PAR, PARAM_DEFAULT_HS_PAR);
00750     retParams.bw = iniParams.getEntryAs<double>(INI_HS_SECTION,
      INI_HS_BW, PARAM_DEFAULT_HS_BW);
00751
00752     return retParams;
00753 }
00754
00755
00763 template<class T>
00764 mdata::Population<T>* Experiment<T>::popPoolRemove()
00765 {
00766     mdata::Population<T>* retPop = nullptr;
00767     std::chrono::microseconds waitTime(10);
00768
00769     while (true)
00770     {
00771         {
00772             std::lock_guard<std::mutex> lk(popPoolMutex);
00773             if (populationsPool.size() > 0)
00774             {
00775                 retPop = populationsPool.back();
00776                 populationsPool.pop_back();
00777             }
00778         }
00779
00780         if (retPop != nullptr)
00781             return retPop;
00782         else
00783             std::this_thread::sleep_for(waitTime);
00784     }
00785 }
00786
00795 template<class T>
00796 void Experiment<T>::popPoolAdd(mdata::Population<T>* popPtr)
00797 {
00798     if (popPtr == nullptr) return;
00799
00800     std::lock_guard<std::mutex> lk(popPoolMutex);
00801
00802     populationsPool.push_back(popPtr);
00803 }
00804
00811 template<class T>
00812 bool Experiment<T>::parseFuncBounds()
00813 {
00814     if (vBounds == nullptr) return false;
00815
00816     const string delim = ",";
00817     const string section = "function_range";
00818     string s_min;
00819     string s_max;
00820
00821     // Extract the bounds for each function
```

```
00822     for (unsigned int i = 1; i <= NUM_FUNCTIONS; i++)
00823     {
00824         // Get bounds entry from ini file for current function
00825         string entry = iniParams.getEntry(section, to_string(i));
00826         if (entry.empty())
00827         {
00828             cerr << "Error parsing bounds for function: " << i << endl;
00829             return false;
00830         }
00831 
00832         // Find index of ',' delimeter in entry string
00833         auto delimPos = entry.find(delim);
00834         if (delimPos == string::npos || delimPos >= entry.length() - 1)
00835         {
00836             cerr << "Error parsing bounds for function: " << i << endl;
00837             return false;
00838         }
00839 
00840         // Split string and extract min/max strings
00841         s_min = entry.substr((size_t)0, delimPos);
00842         s_max = entry.substr(delimPos + 1, entry.length());
00843         util::s_trim(s_min);
00844         util::s_trim(s_max);
00845 
00846         // Attempt to parse min and max strings into double values
00847         try
00848         {
00849             RandomBounds<T>& b = vBounds[i - 1];
00850             b.min = atof(s_min.c_str());
00851             b.max = atof(s_max.c_str());
00852         }
00853         catch(const std::exception& e)
00854         {
00855             cerr << "Error parsing bounds for function: " << i << endl;
00856             std::cerr << e.what() << '\n';
00857             return false;
00858         }
00859     }
00860 
00861     return true;
00862 }
00863 
00871 template<class T>
00872 bool Experiment<T>::allocatePopulationPool(size_t count, size_t
     popSize, size_t dimensions)
00873 {
00874     releasePopulationPool();
00875 
00876     std::lock_guard<std::mutex> lk(popPoolMutex);
00877 
00878     try
00879     {
00880         for (int i = 0; i < count; i++)
00881         {
00882             auto newPop = new(std::nothrow) mdata::Population<T>(popSize, dimensions);
00883             if (newPop == nullptr)
00884             {
00885                 std::cerr << "Error allocating populations." << '\n';
00886                 return false;
00887             }
00888 
00889             populationsPool.push_back(newPop);
00890         }
00891 
00892         return true;
00893     }
00894     catch(const std::exception& e)
00895     {
00896         std::cerr << e.what() << '\n';
00897         return false;
00898     }
00899 }
00900 
00904 template<class T>
00905 void Experiment<T>::releasePopulationPool()
00906 {
00907     std::lock_guard<std::mutex> lk(popPoolMutex);
00908 
00909     if (populationsPool.size() == 0) return;
00910 
00911     for (int i = 0; i < populationsPool.size(); i++)
00912     {
00913         if (populationsPool[i] != nullptr)
00914         {
00915             delete populationsPool[i];
00916             populationsPool[i] = nullptr;
00917         }
```

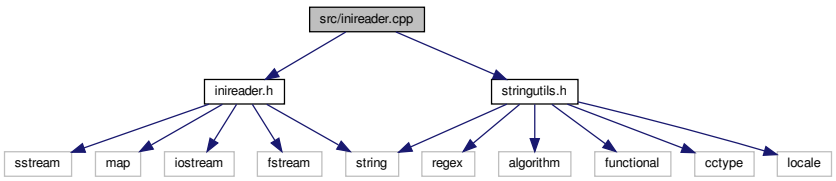
```
00918     }
00919
00920     populationsPool.clear();
00921 }
00922
00930 template<class T>
00931 bool Experiment<T>::allocateVBounds()
00932 {
00933     vBounds = util::allocArray<RandomBounds<T>>(NUM_FUNCTIONS);
00934     return vBounds != nullptr;
00935 }
00936
00940 template<class T>
00941 void Experiment<T>::releaseVBounds()
00942 {
00943     if (vBounds == nullptr) return;
00944
00945     util::releaseArray<RandomBounds<T>>(vBounds);
00946 }
00947
00956 template<class T>
00957 bool Experiment<T>::allocateThreadPool(size_t numThreads)
00958 {
00959     releaseThreadPool();
00960
00961     tPool = new(std::nothrow) ThreadPool(numThreads);
00962     return tPool != nullptr;
00963 }
00964
00968 template<class T>
00969 void Experiment<T>::releaseThreadPool()
00970 {
00971     if (tPool == nullptr) return;
00972
00973     delete tPool;
00974     tPool = nullptr;
00975 }
00976
00977 // Explicit template specializations due to separate implementations in this CPP file
00978 template class mfunc::Experiment<float>;
00979 template class mfunc::Experiment<double>;
00980 template class mfunc::Experiment<long double>;
00981
00982 // ==========================
00983 // End of experiment.cpp
00984 // ==========================
```

6.27 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:



6.27.1 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

Definition in file inireader.cpp.

6.28 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
00072     return it->second.find(entry) != it->second.end();
00073 }
00074
00084 std::string IniReader::getEntry(std::string section, std::string entry, std::string
      defVal)
00085 {
00086     if (!entryExists(section, entry)) return defVal;
00087
00088     return iniMap[section][entry];
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 // ========================
```

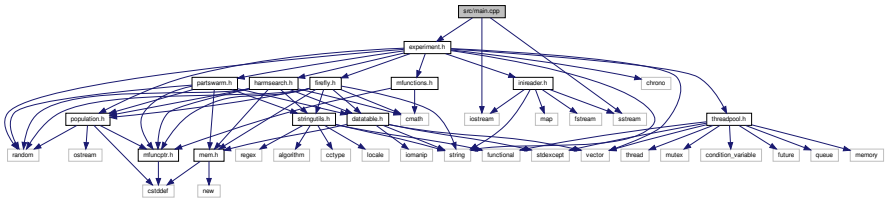
6.29 src/main.cpp File Reference

Program entry point. Creates and runs CS471 project 4 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)

6.29.1 Detailed Description

Program entry point. Creates and runs CS471 project 4 experiment.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.4

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file main.cpp.

6.29.2 Function Documentation

6.29.2.1 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
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 }
```

6.29.2.2 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

T	

Parameters

paramFile	

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 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 }
```

6.30 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 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;
```

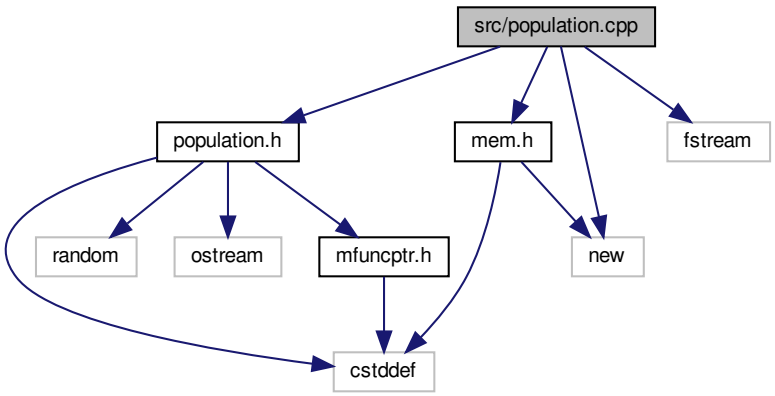
```
00072     }
00073
00074     // Run experiment with correct data type and return success code
00075     switch (dataType)
00076     {
00077        case 0:
00078           return runExp<float>(argv[1]);
00079        case 1:
00080           return runExp<double>(argv[1]);
00081        case 2:
00082           return runExp<long double>(argv[1]);
00083        default:
00084           return EXIT_FAILURE;
00085     }
00086 }
00087
00088 // ========================
00089 // End of main.cpp
00090 // ========================
```

6.31 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 <fstream>
```

Include dependency graph for population.cpp:



6.31.1 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

Definition in file population.cpp.

6.32 population.cpp

```
00001
00012 #include "population.h"
00013 #include "mem.h"
00014 #include <new>
00015 #include <fstream>
00016
00017 using namespace mdata;
00018 using namespace util;
00019
00027 template <class T>
00028 Population<T>::Population(size_t pSize, size_t dimensions)
00029     : popMatrix(nullptr), popSize(pSize), popDim(dimensions), rdev(), rgen(rdev())
00030 {
00031     if (!allocPopMatrix() || !allocPopFitness())
00032         throw std::bad_alloc();
00033 }
00034
00040 template <class T>
00041 Population<T>::~Population()
00042 {
00043     releasePopMatrix();
00044     releasePopFitness();
00045 }
00046
00054 template <class T>
00055 bool Population<T>::isReady()
00056 {
00057     return popMatrix != nullptr && popFitness != nullptr;
00058 }
00059
00066 template <class T>
00067 size_t Population<T>::getPopulationSize()
00068 {
00069     return popSize;
00070 }
00071
00078 template <class T>
00079 size_t Population<T>::getDimensionsSize()
00080 {
00081     return popDim;
00082 }
00083
00091 template <class T>
00092 T* Population<T>::getPopulationPtr(size_t popIndex)
00093 {
00094     if (popFitness == nullptr || popIndex >= popSize) return nullptr;
00095
00096     return popMatrix[popIndex];
00097 }
00098
00099 template <class T>
00100 T* Population<T>::getBestPopulationPtr()
00101 {
00102     return getPopulationPtr(getBestFitnessIndex());
00103 }
00104
00115 template <class T>
00116 bool Population<T>::generate(T minBound, T maxBound)
```

```
00117 {
00118     if (popMatrix == nullptr) return false;
00119 
00120     // Set up a uniform distribution for the random number generator with the correct function bounds
00121     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00122 
00123     // Generate values for all vectors in popMatrix
00124     for (size_t s = 0; s < popSize; s++)
00125     {
00126         for (size_t d = 0; d < popDim; d++)
00127         {
00128             T rand = (T)dist(rgen);
00129             popMatrix[s][d] = rand;
00130         }
00131     }
00132 
00133     // Reset popFitness values to 0
00134     initArray<T>(popFitness, popSize, (T)0.0);
00135 
00136     return true;
00137 }
00138 
00139 template <class T>
00140 bool Population<T>::generateSingle(size_t popIndex, T minBound, T maxBound)
00141 {
00142     if (popMatrix == nullptr || popIndex >= popSize) return false;
00143 
00144     // Set up a uniform distribution for the random number generator with the correct function bounds
00145     std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);
00146 
00147     for (size_t d = 0; d < popDim; d++)
00148     {
00149         T rand = (T)dist(rgen);
00150         popMatrix[popIndex][d] = rand;
00151     }
00152 
00153     popFitness[popIndex] = 0;
00154 
00155     return true;
00156 }
00157 
00166 template<class T>
00167 bool Population<T>::setFitness(size_t popIndex, T value)
00168 {
00169     if (popFitness == nullptr || popIndex >= popSize) return false;
00170 
00171     popFitness[popIndex] = value;
00172 
00173     return true;
00174 }
00175 
00186 template<class T>
00187 bool Population<T>::calcFitness(size_t popIndex,
      mfunc::mfuncPtr<T> funcPtr)
00188 {
00189     if (popFitness == nullptr || popIndex >= popSize) return false;
00190 
00191     popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);
00192 
00193     return true;
00194 }
00195 
00196 template<class T>
00197 bool Population<T>::calcAllFitness(
      mfunc::mfuncPtr<T> funcPtr)
00198 {
00199     for (size_t i = 0; i < popSize; i++)
00200     {
00201         if (!calcFitness(i, funcPtr))
00202             return false;
00203     }
00204 
00205     return true;
00206 }
00207 
00215 template<class T>
00216 T Population<T>::getFitness(size_t popIndex)
00217 {
00218     if (popFitness == nullptr || popIndex >= popSize) return 0;
00219 
00220     return popFitness[popIndex];
00221 }
00222 
00230 template<class T>
00231 T* Population<T>::getFitnessPtr(size_t popIndex)
00232 {
00233     if (popFitness == nullptr || popIndex >= popSize) return 0;
```

```
00234
00235     return &popFitness[popIndex];
00236 }
00237
00244 template<class T>
00245 T* Population<T>::getBestFitnessPtr()
00246 {
00247     return &popFitness[getBestFitnessIndex()];
00248 }
00249
00256 template<class T>
00257 size_t Population<T>::getBestFitnessIndex()
00258 {
00259     size_t bestIndex = 0;
00260
00261     for (size_t i = 1; i < popSize; i++)
00262     {
00263         if (popFitness[i] < popFitness[bestIndex])
00264             bestIndex = i;
00265     }
00266
00267     return bestIndex;
00268 }
00269
00270 template<class T>
00271 T Population<T>::getBestFitness()
00272 {
00273     return getFitness(getBestFitnessIndex());
00274 }
00275
00276 template<class T>
00277 size_t Population<T>::getWorstFitnessIndex()
00278 {
00279     size_t worstIndex = 0;
00280
00281     for (size_t i = 1; i < popSize; i++)
00282     {
00283         if (popFitness[i] > popFitness[worstIndex])
00284             worstIndex = i;
00285     }
00286
00287     return worstIndex;
00288 }
00289
00290 template<class T>
00291 T Population<T>::getWorstFitness()
00292 {
00293     return getFitness(getWorstFitnessIndex());
00294 }
00295
00296 template<class T>
00297 void Population<T>::sortFitnessAscend()
00298 {
00299     qs_fit_ascend(0, popSize - 1);
00300 }
00301
00302 template<class T>
00303 void Population<T>::sortFitnessDescend()
00304 {
00305     qs_fit_descend(0, popSize - 1);
00306 }
00307
00308 template<class T>
00309 bool Population<T>::copyFrom(Population<T>* srcPtr, size_t srcIndex,
      size_t destIndex)
00310 {
00311     if (srcPtr == nullptr) return false;
00312
00313     const size_t srcDim = srcPtr->getDimensionsSize();
00314     if (srcDim != popDim) return false;
00315
00316     T* srcVector = srcPtr->getPopulationPtr(srcIndex);
00317     T* destVector = getPopulationPtr(destIndex);
00318
00319     if (srcVector == nullptr || destVector == nullptr) return false;
00320
00321     copyArray<T>(srcVector, destVector, popDim);
00322     setFitness(destIndex, srcPtr->getFitness(srcIndex));
00323
00324     return true;
00325 }
00326
00327 template<class T>
00328 bool Population<T>::copyAllFrom(Population<T>* srcPtr)
00329 {
00330     if (srcPtr == nullptr) return false;
00331
```

```
00332     const size_t srcSize = srcPtr->getPopulationSize();
00333     const size_t srcDim = srcPtr->getDimensionsSize();
00334
00335     if (srcSize != popSize || srcDim != popDim)
00336         return false;
00337
00338     for (size_t i = 0; i < popSize; i++)
00339     {
00340         if (!copyFrom(srcPtr, i, i))
00341             return false;
00342     }
00343
00344     return true;
00345 }
00346
00347 template<class T>
00348 bool Population<T>::copyPopulation(T* src, size_t destIndex)
00349 {
00350     T* destVect = getPopulationPtr(destIndex);
00351     if (destVect == nullptr)
00352         return false;
00353
00354     for (size_t i = 0; i < popDim; i++)
00355     {
00356         destVect[i] = src[i];
00357     }
00358
00359     return true;
00360 }
00361
00370 template<class T>
00371 void Population<T>::outputPopulation(std::ostream& outStream, const char*
      delim, const char* lineBreak)
00372 {
00373     if (popMatrix == nullptr) return;
00374
00375     for (size_t j = 0; j < popSize; j++)
00376     {
00377         for (size_t k = 0; k < popDim; k++)
00378         {
00379             outStream << popMatrix[j][k];
00380             if (k < popDim - 1)
00381                 outStream << delim;
00382         }
00383
00384         outStream << lineBreak;
00385     }
00386 }
00387
00388 template<class T>
00389 bool Population<T>::outputPopulationCsv(std::string filePath)
00390 {
00391     static const char* delim = ",";
00392     static const char* newline = "\n";
00393
00394     std::ofstream file;
00395     file.open(filePath, std::ios::out | std::ios::trunc);
00396     if (!file.good()) return false;
00397
00398     outputPopulation(file, delim, newline);
00399     file.close();
00400
00401     return true;
00402 }
00403
00412 template<class T>
00413 void Population<T>::outputFitness(std::ostream& outStream, const char* delim,
      const char* lineBreak)
00414 {
00415     if (popFitness == nullptr) return;
00416
00417     for (size_t j = 0; j < popSize; j++)
00418     {
00419         outStream << popFitness[j];
00420             if (j < popSize - 1)
00421                 outStream << delim;
00422     }
00423
00424     if (lineBreak != nullptr)
00425         outStream << lineBreak;
00426 }
00427
00434 template <class T>
00435 bool Population<T>::allocPopMatrix()
00436 {
00437     if (popSize == 0 || popDim == 0) return false;
00438
```

```
00439     popMatrix = allocMatrix<T>(popSize, popDim);
00440     initMatrix<T>(popMatrix, popSize, popDim, 0);
00441
00442     return popMatrix != nullptr;
00443 }
00444
00450 template <class T>
00451 void Population<T>::releasePopMatrix()
00452 {
00453     releaseMatrix<T>(popMatrix, popSize);
00454 }
00455
00462 template <class T>
00463 bool Population<T>::allocPopFitness()
00464 {
00465     if (popSize == 0 || popDim == 0) return false;
00466
00467     popFitness = allocArray<T>(popSize);
00468     initArray<T>(popFitness, popSize, 0);
00469
00470     return popFitness != nullptr;
00471 }
00472
00478 template <class T>
00479 void Population<T>::releasePopFitness()
00480 {
00481     releaseArray<T>(popFitness);
00482 }
00483
00484 // ========================================
00485 // Quicksort Implementation modified from:
00486 // https://www.geeksforgeeks.org/quick-sort/
00487 // ========================================
00488
00489 template <class T>
00490 void Population<T>::qs_swapval(T& a, T& b)
00491 {
00492     T t = a;
00493     a = b;
00494     b = t;
00495 }
00496
00497 template <class T>
00498 void Population<T>::qs_swapptr(T*& a, T*& b)
00499 {
00500     T* t = a;
00501     a = b;
00502     b = t;
00503 }
00504
00505 template <class T>
00506 long Population<T>::part_fit_ascend(long low, long high)
00507 {
00508     T pivot = popFitness[high]; // pivot
00509     long i = (low - 1);  // Index of smaller element
00510
00511     for (long j = low; j <= high- 1; j++)
00512     {
00513         if (popFitness[j] <= pivot)
00514         {
00515             i++;    // increment index of smaller element
00516             qs_swapval(popFitness[i], popFitness[j]);
00517             qs_swapptr(popMatrix[i], popMatrix[j]);
00518         }
00519     }
00520     qs_swapval(popFitness[i + 1], popFitness[high]);
00521     qs_swapptr(popMatrix[i + 1], popMatrix[high]);
00522
00523     return (i + 1);
00524 }
00525
00526 template <class T>
00527 void Population<T>::qs_fit_ascend(long low, long high)
00528 {
00529     if (low < high)
00530     {
00531         long pi = part_fit_ascend(low, high);
00532
00533         // Separately sort elements before
00534         // partition and after partition
00535         qs_fit_ascend(low, pi - 1);
00536         qs_fit_ascend(pi + 1, high);
00537     }
00538 }
00539
00540 template <class T>
00541 long Population<T>::part_fit_descend(long low, long high)
```

```
00542 {
00543     T pivot = popFitness[high]; // pivot
00544     long i = (low - 1);  // Index of smaller element
00545 
00546     for (long j = low; j <= high- 1; j++)
00547     {
00548         if (popFitness[j] > pivot)
00549         {
00550             i++;    // increment index of smaller element
00551             qs_swapval(popFitness[i], popFitness[j]);
00552             qs_swapptr(popMatrix[i], popMatrix[j]);
00553         }
00554     }
00555     qs_swapval(popFitness[i + 1], popFitness[high]);
00556     qs_swapptr(popMatrix[i + 1], popMatrix[high]);
00557 
00558     return (i + 1);
00559 }
00560 
00561 template <class T>
00562 void Population<T>::qs_fit_descend(long low, long high)
00563 {
00564     if (low < high)
00565     {
00566         long pi = part_fit_descend(low, high);
00567 
00568         // Separately sort elements before
00569         // partition and after partition
00570         qs_fit_descend(low, pi - 1);
00571         qs_fit_descend(pi + 1, high);
00572     }
00573 }
00574 
00575 // Explicit template specializations due to separate implementations in this CPP file
00576 template class mdata::Population<float>;
00577 template class mdata::Population<double>;
00578 template class mdata::Population<long double>;
00579 
00580 // ========================
00581 // End of population.cpp
00582 // ========================
```

Index