CS471 Project 2

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

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 enums Namespace Reference

Classes

- struct AlgorithmNames

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

Enumerations

- enum Algorithm { BlindSearch = 0, LocalSearch = 1, Count = 2 }

    Enum of different available search algorithms.

5.1.1 Enumeration Type Documentation

5.1.1.1 Algorithm

`enum enums::Algorithm`

Enum of different available search algorithms.

Enumerator

BlindSearch	
LocalSearch	
Count	

Definition at line 28 of file searchalg.h.

```
00029     {
        BlindSearch = 0,
        LocalSearch = 1,
        Count = 2
    };
```

5.2 mdata Namespace Reference

Classes

- class BlindSearch

  The BlindSearch class implements the Blind Search algorithm, which is ran using the overridden SearchAlgorithm::run() function.
- class DataTable

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

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

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

  The SearchAlgorithm class is used as a base class for other implemented search algorithms. Provides a common interface to run each algorithm.
- struct TestParameters

  Packs together various test experiment parameters.
- struct TestResult

5.3 mfunc Namespace Reference

Classes

- class Experiment

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

  get() returns a function's description Returns a C-string description for the given function id if the id is valid. Otherwise returns null
- struct Functions

  Struct containing all static math functions. A function can be called directly by name, or indirectly using Functions::get or Functions::exec.
- struct RandomBounds

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

Typedefs

- template<class T >
  using mfuncPtr = T(∗)(T ∗, size_t)

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

Variables

- constexpr const unsigned int NUM_FUNCTIONS = 18

5.3.1 Detailed Description

Scope for all math functions

5.3.2 Typedef Documentation

5.3.2.1 mfuncPtr

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

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

Template Parameters

T	Data type for vector and return value

Definition at line 28 of file mfuncptr.h.

5.3.3 Variable Documentation

5.3.3.1 NUM_FUNCTIONS

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

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

Definition at line 47 of file mfunctions.h.

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

5.4 util Namespace Reference

Classes

- class IniReader

  The IniReader class is a simple ∗.ini file reader and parser.

Functions

- template<class T = double>
  void initArray (T ∗a, size_t size, T val)

  Initializes an array with some set value.
- template<class T = double>
  void initMatrix (T ∗∗m, size_t rows, size_t cols, T val)

  Initializes a matrix with a set value for each entry.
- template<class T = double>
  void releaseArray (T ∗&a)

  Releases an allocated array's memory and sets the pointer to nullptr.
- template<class T = double>
  void releaseMatrix (T ∗∗&m, size_t rows)

  Releases an allocated matrix's memory and sets the pointer to nullptr.
- template<class T = double>
  T ∗ allocArray (size_t size)

  Allocates a new array of the given data type.
- template<class T = double>
  T ∗∗ allocMatrix (size_t rows, size_t cols)

  Allocates a new matrix of the given data type.
- template<class T = double>
  void copyArray (T ∗src, T ∗dest, size_t size)

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

5.4.1 Function Documentation

5.4.1.1 allocArray()

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

Allocates a new array of the given data type.

Template Parameters

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 108 of file mem.h.

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

5.4.1.2 allocMatrix()

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

Allocates a new matrix of the given data type.

Template Parameters

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 122 of file mem.h.

```
{
    T** m = (T**)allocArray<T*>(rows);
    if (m == nullptr) return nullptr;

    for (size_t i = 0; i < rows; i++)
    {
        m[i] = allocArray<T>(cols);
        if (m[i] == nullptr)
        {
            releaseMatrix<T>(m, rows);
            return nullptr;
        }
    }

    return m;
}
```

5.4.1.3 copyArray()

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

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

Template Parameters

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 149 of file mem.h.

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

5.4.1.4 initArray()

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

Initializes an array with some set value.

Template Parameters

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().

```
{
    if (a == nullptr) return;

    for (size_t i = 0; i < size; i++)
    {
        a[i] = val;
    }
}
```

5.4.1.5 initMatrix()

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

Initializes a matrix with a set value for each entry.

Template Parameters

Data	type of matrix entries

Parameters

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().

```
{
    if (m == nullptr) return;

    for (size_t i = 0; i < rows; i++)
    {
        initArray(m[i], cols, val);
    }
}
```

5.4.1.6 releaseArray()

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

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

Template Parameters

Data	type of array

Parameters

a	Pointer to array

Definition at line 66 of file mem.h.

```
{
    if (a == nullptr) return;

    delete[] a;
    a = nullptr;
}
```

5.4.1.7 releaseMatrix()

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

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

Template Parameters

Data	type of the matrix

Parameters

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

Definition at line 82 of file mem.h.

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

```
{
    if (m == nullptr) return;

    for (size_t i = 0; i < rows; i++)
    {
        if (m[i] != nullptr)
        {
            // Release each row
            releaseArray<T>(m[i]);
        }
    }

    // Release columns
    delete[] m;
    m = nullptr;
}
```

Chapter 6

Class Documentation

6.1 enums::AlgorithmNames Struct Reference

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

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

Static Public Member Functions

- static const char ∗ get (Algorithm alg)

Static Public Attributes

- static constexpr const char ∗ BLIND_SEARCH = "Blind Search"
- static constexpr const char ∗ LOCAL_SEARCH = "Local Search"

6.1.1 Detailed Description

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

Definition at line 39 of file searchalg.h.

6.1.2 Member Function Documentation

6.1.2.1 get()

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

Definition at line 44 of file searchalg.h.

References enums::BlindSearch, and enums::LocalSearch.

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

```
{
    switch (alg)
    {
        case Algorithm::BlindSearch:
            return BLIND_SEARCH;
        case Algorithm::LocalSearch:
            return LOCAL_SEARCH;
        default:
            return "";
            break;
    }
}
```

6.1.3 Member Data Documentation

6.1.3.1 BLIND_SEARCH

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

Definition at line 41 of file searchalg.h.

6.1.3.2 LOCAL_SEARCH

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

Definition at line 42 of file searchalg.h.

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

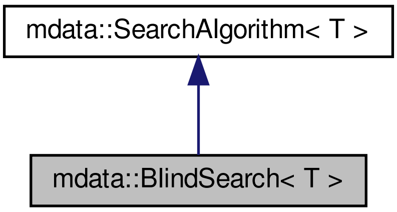
- include/searchalg.h

6.2 mdata::BlindSearch< T > Class Template Reference

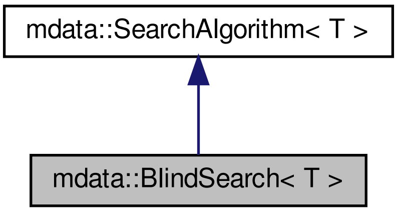
The BlindSearch class implements the Blind Search algorithm, which is ran using the overridden SearchAlgorithm↩ ::run() function.

`#include <blindsearch.h>`

Inheritance diagram for mdata::BlindSearch< T >:



Collaboration diagram for mdata::BlindSearch< T >:



Public Member Functions

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

  Executes Blind Search with the given population and parameters.

Additional Inherited Members

6.2.1 Detailed Description

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

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

Template Parameters

T	Data type used

Definition at line 27 of file blindsearch.h.

6.2.2 Member Function Documentation

6.2.2.1 run()

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

Executes Blind Search with the given population and parameters.

Parameters

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

Returns

TestResult<T> Returns a TestResult struct containing the error code, fitness, and execution time

Implements mdata::SearchAlgorithm< T >.

Definition at line 44 of file blindsearch.h.

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

```
    {
        // Get population size and dimensions
        size_t popSize = pop->getPopulationSize();
        size_t dimSize = pop->getDimensionsSize();

        // Make sure funcPtr is valid;
        if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with error code 1

        // Start recording execution time
        startTimer();
```

```

            // Generate values for population vector matrix
            pop->generate(fMin, fMax);

            // For each population vector, calculate the fitness using the funcPtr
            for (size_t sol = 0; sol < popSize; sol++)
            {
                // Populate fitness values using given math function pointer
                if (!pop->calcFitness(sol, funcPtr))
                    return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
            }

            // Return best fitness value in population
            return TestResult<T>(0, *pop->getBestFitnessPtr(), stopTimer());
        }
```

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

- include/blindsearch.h

6.3 mdata::DataTable< T > Class Template Reference

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

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

Public Member Functions

- DataTable (size_t _rows, size_t _cols)

  Construct a new Data Table object Throws std::length_error and std::bad_alloc.
- ∼DataTable ()

  Destroy the Data Table object.
- std::string getColLabel (size_t colIndex)

  Gets the string label for the column with the given index.
- void setColLabel (size_t colIndex, std::string newLabel)

  Sets the string label for the column with the given index.
- T getEntry (size_t row, size_t col)

  Returns the value in the table at the given row and column.
- void setEntry (size_t row, size_t col, T val)

  Set the value for the table entry at the given row and column.
- bool exportCSV (const char *filePath)

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

6.3.1 Detailed Description

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

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

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

Set a column's label:

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

Set an entry in the table:

table.setEntry(n, m, value);

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

Export the table to a *.csv file:

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

Definition at line 50 of file datatable.h.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 DataTable()

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

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

Parameters

_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         }
```

6.3.2.2 ∼DataTable()

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

Destroy the Data Table object.

Definition at line 77 of file datatable.h.

References util::releaseMatrix().

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

6.3.3 Member Function Documentation

6.3.3.1 exportCSV()

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

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

Parameters

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 155 of file datatable.h.

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

```
        {
            if (dataMatrix == nullptr) return false;

            using namespace std;
            ofstream outFile;
            outFile.open(filePath, ofstream::out | ofstream::trunc);
            if (!outFile.good()) return false;

            // Print column labels
            for (unsigned int c = 0; c < cols; c++)
            {
                outFile << colLabels[c];
                if (c < cols - 1) outFile << ",";
            }

            outFile << endl;

            // Print data rows
            for (unsigned int r = 0; r < rows; r++)
            {
                for (unsigned int c = 0; c < cols; c++)
                {
                    outFile << std::setprecision(8) << dataMatrix[r][c];
                    if (c < cols - 1) outFile << ",";
                }
                outFile << endl;
            }

            outFile.close();
            return true;
        }
```

6.3.3.2 getColLabel()

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

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

Parameters

colIndex	Index of the column

Returns

std::string String value of the column label

Definition at line 88 of file datatable.h.

```
        {
            if (colIndex >= colLabels.size())
                throw std::out_of_range("Column index out of range");

            return colLabels[colIndex];
        }
```

6.3.3.3 getEntry()

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

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

Parameters

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 117 of file datatable.h.

```
        {
            if (dataMatrix == nullptr)
                throw std::runtime_error("Data matrix not allocated");
            if (row >= rows)
                throw std::out_of_range("Table row out of range");
            else if (col >= cols)
                throw std::out_of_range("Table column out of range");

            return dataMatrix[row][col];
        }
```

6.3.3.4 setColLabel()

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

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

Parameters

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

Definition at line 102 of file datatable.h.

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

```
        {
             if (colIndex >= colLabels.size())
                throw std::out_of_range("Column index out of range");

            colLabels[colIndex] = newLabel;
        }
```

6.3.3.5 setEntry()

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

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

Parameters

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

Definition at line 136 of file datatable.h.

```
        {
            if (dataMatrix == nullptr)
                throw std::runtime_error("Data matrix not allocated");
            if (row >= rows)
                throw std::out_of_range("Table row out of range");
            else if (col >= cols)
                throw std::out_of_range("Table column out of range");

            dataMatrix[row][col] = val;
        }
```

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

- include/datatable.h

6.4 mfunc::Experiment< T > Class Template Reference

Contains classes for running the CS471 project experiment.

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

Public Member Functions

- Experiment ()

  Construct a new Experiment object.
- ∼Experiment ()

  Destroys the Experiment object.
- bool init (const char ∗paramFile)

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

  Executes all functions as specified in the CS471 project 2 document, records results, and outputs the data as a ∗.csv file.
- int testFuncThreaded (mdata::TestParameters< T > tParams)

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

6.4.1 Detailed Description

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

Contains classes for running the CS471 project experiment.

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

Definition at line 52 of file experiment.h.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 Experiment()

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

Construct a new Experiment object.

Definition at line 43 of file experiment.cpp.

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

6.4.2.2 ∼Experiment()

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

Destroys the Experiment object.

Definition at line 53 of file experiment.cpp.

```
{
    releaseThreadPool();
    releasePopulationPool();
    releaseVBounds();
}
```

6.4.3 Member Function Documentation

6.4.3.1 init()

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

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

Parameters

paramFile	File path to the parameter ini file

Returns

Returns true if initialization was successful. Otherwise false.

Definition at line 69 of file experiment.cpp.

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

Referenced by runExp().

```
{
    try
    {
        // Open and parse parameters file
        if (!iniParams.openFile(paramFile))
        {
            cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
            return false;
        }

        // Extract test parameters from ini file
        long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_POPULATION);
        long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_DIMENSIONS);
        long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_ITERATIONS);
        long numberThreads = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_NUMTHREADS);
        alpha = iniParams.getEntryAs<T>(INI_TEST_SECTION, INI_TEST_ALPHA);
        unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(INI_TEST_SECTION, INI_TEST_ALGORITHM);
        resultsFile = iniParams.getEntry(INI_TEST_SECTION, INI_TEST_RESULTSFILE);
        execTimesFile = iniParams.getEntry(INI_TEST_SECTION, INI_TEST_EXECTIMESFILE);

        // Verify test parameters
        if (numberSol <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_POPULATION << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberDim <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberIter <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ITERATIONS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberThreads <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (alpha == 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ALPHA << " is missing or is equal to zero: " << paramFile << endl;
            return false;
        }
        else if (selectedAlg >= static_cast<unsigned int>(enums::Algorithm::Count))
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }

        // Cast iterations and test algorithm to correct types
        iterations = (size_t)numberIter;
        testAlg = static_cast<enums::Algorithm>(selectedAlg);

        // Print test parameters to console
        cout << "Population size: " << numberSol << endl;
        cout << "Dimensions: " << numberDim << endl;
        cout << "Iterations: " << iterations << endl;
        cout << "Alpha value: " << alpha << endl;
        cout << "Algorithm: " << enums::AlgorithmNames::get(testAlg) << endl;

        // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
        if (!allocatePopulationPool((size_t)numberThreads, (size_t)numberSol, (size_t)numberDim))
        {
            cerr << "Experiment init failed: Unable to allocate populations." << endl;
            return false;
        }

        // Allocate memory for function vector bounds
        if (!allocateVBounds())
        {
            cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
            return false;
        }
```

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

6.4.3.2 testAllFunc()

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

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

Returns

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

Definition at line 191 of file experiment.cpp.

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

Referenced by runExp().

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

```

    // Start recording total execution time
    high_resolution_clock::time_point t_start = high_resolution_clock::now();

    // For each of the NUM_FUNCTIONS functions, prepare a TestParameters
    // struct and queue an asynchronous test that will be picked up and
    // executed by one of the threads in the thread pool.
    for (unsigned int i = 0; i < NUM_FUNCTIONS; i++)
    {
        // Update column labels for results and exec times tables
        resultsTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
        execTimesTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));

        // Queue up a new function test for each iteration
        for (size_t iter = 0; iter < iterations; iter++)
        {
            mdata::TestParameters<T> curParam;
            curParam.funcId = i + 1;
            curParam.alpha = alpha;
            curParam.alg = testAlg;
            curParam.resultsTable = &resultsTable;
            curParam.execTimesTable = &execTimesTable;
            curParam.resultsCol = i;
            curParam.execTimesCol = i;
            curParam.resultsRow = iter;
            curParam.execTimesRow = iter;

            // Add function test to async queue
            testFutures.emplace_back(
                tPool->enqueue(&Experiment<T>::testFuncThreaded, this, curParam)
            );
        }
    }

    // Get the total number of async tasks queued
    const double totalFutures = static_cast<double>(testFutures.size());
    int tensPercentile = -1;
    std::chrono::microseconds waitTime(100);

    // Loop until all async tasks are completed and the thread futures
    // array is empty
    while (testFutures.size() > 0)
    {
        // Sleep a little bit since the async thread tasks are higher priority
        std::this_thread::sleep_for(waitTime);

        // Get iterator to first thread future
        auto it = testFutures.begin();

        // Loop through all thread futures
        while (it != testFutures.end())
        {
            if (!it->valid())
            {
                // An error occured with one of the threads
                cerr << "Error: Thread future invalid.";
                tPool->stopAndJoinAll();
                return 1;
            }

            // Get the status of the current thread future (async task)
            std::future_status status = it->wait_for(waitTime);
            if (status == std::future_status::ready)
            {
                // Task has completed, get return value
                int errCode = it->get();
                if (errCode)
                {
                    // An error occurred while running the task.
                    // Bail out of function
                    tPool->stopAndJoinAll();
                    return errCode;
                }

                // Remove processed task future from vector
                it = testFutures.erase(it);

                // Calculate the percent completed of all tasks, rounded to the nearest 10%
                int curPercentile = static_cast<int>(((totalFutures - testFutures.size()) / totalFutures) * 10);
                if (curPercentile > tensPercentile)
                {
                    // Print latest percent value to the console
                    tensPercentile = curPercentile;
                    cout << "~" << (tensPercentile * 10) << "% " << flush;
                }
            }
            else
            {
                // Async task has not yet completed, advance to the next one
                it++;
            }
        }
    }

    // Record total execution time and print it to the console
    high_resolution_clock::time_point t_end = high_resolution_clock::now();
    long double totalExecTime = static_cast<long double>(duration_cast<nanoseconds>(t_end - t_start).count()) / 1000000000.0L;

    cout << endl << "Test finished. Total time: " << std::setprecision(7) << totalExecTime << " seconds." << endl;

    if (!resultsFile.empty())
    {
        // Export results table to a *.csv file
        cout << "Exporting results to: " << resultsFile << endl;
        resultsTable.exportCSV(resultsFile.c_str());
    }

    if (!execTimesFile.empty())
    {
        // Export exec times table to a *.csv file
        cout << "Exporting execution times to: " << execTimesFile << endl;
        execTimesTable.exportCSV(execTimesFile.c_str());
    }

    cout << flush;

    return 0;
}
```

6.4.3.3 testFuncThreaded()

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

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

Template Parameters

T	The data type used by the test

Parameters

tParams	The parameters used to set up the test

Returns

int An error code if any

Definition at line 329 of file experiment.cpp.

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

```
{
    mdata::SearchAlgorithm<T>* alg;

    // Construct a search algorithm object for the selected alg
    switch (tParams.alg)
    {
        case enums::Algorithm::BlindSearch:
            alg = new mdata::BlindSearch<T>();
            break;
        case enums::Algorithm::LocalSearch:
            alg = new mdata::LocalSearch<T>();
            break;
        default:
            cerr << "Invalid algorithm selected." << endl;
            return 1;
    }

    // Retrieve the function bounds
    const RandomBounds<T>& funcBounds = vBounds[tParams.funcId - 1];

    // Retrieve the next available population object from the population pool
    mdata::Population<T>* pop = popPoolRemove();

    // Run the search algorithm one and record the results
    auto tResult = alg->run(Functions<T>::get(tParams.funcId), funcBounds.min, funcBounds.max, pop, tParams.alpha);

    // Place the population object back into the pool to be reused by anther thread
    popPoolAdd(pop);

    if (tResult.err)
    {
        cerr << "Error while testing function " << tParams.funcId << endl;
        return tResult.err;
    }

    // Update results table and execution times table with algorithm results
    tParams.resultsTable->setEntry(tParams.resultsRow, tParams.resultsCol, tResult.fitness);
    tParams.execTimesTable->setEntry(tParams.execTimesRow, tParams.execTimesCol, tResult.execTime);

    delete alg;
    return 0;
}
```

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

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

6.5 mfunc::FunctionDesc Struct Reference

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

`#include <mfunctions.h>`

Static Public Member Functions

- static const char ∗ get (unsigned int f)

6.5.1 Detailed Description

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

Parameters

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 56 of file mfunctions.h.

6.5.2 Member Function Documentation

6.5.2.1 get()

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

Definition at line 58 of file mfunctions.h.

References _ackleysOneDesc, _ackleysTwoDesc, _alpineDesc, _dejongDesc, _eggHolderDesc, _griewangkDesc, _levyDesc, _mastersCosineWaveDesc, _michalewiczDesc, _pathologicalDesc, _quarticDesc, _ranaDesc, _rastriginDesc, _rosenbrokDesc, _schwefelDesc, _sineEnvelopeSineWaveDesc, _stepDesc, and _stretchedVSineWaveDesc.

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

```
{
    switch (f)
    {
        case 1:
            return _schwefelDesc;
        case 2:
            return _dejongDesc;
        case 3:
            return _rosenbrokDesc;
        case 4:
            return _rastriginDesc;
        case 5:
            return _griewangkDesc;
        case 6:
            return _sineEnvelopeSineWaveDesc;
        case 7:
            return _stretchedVSineWaveDesc;
        case 8:
            return _ackleysOneDesc;
        case 9:
            return _ackleysTwoDesc;
        case 10:
            return _eggHolderDesc;
        case 11:
            return _ranaDesc;
        case 12:
            return _pathologicalDesc;
        case 13:
            return _michalewiczDesc;
        case 14:
            return _mastersCosineWaveDesc;
        case 15:
            return _quarticDesc;
        case 16:
            return _levyDesc;
        case 17:
            return _stepDesc;
        case 18:
            return _alpineDesc;
        default:
            return NULL;
    }
}
```

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

- include/mfunctions.h

6.6 mfunc::Functions< T > Struct Template Reference

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

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

Static Public Member Functions

- static T schwefel (T ∗v, size_t n)

  Function 1. Implementation of Schwefel's mathematical function.
- static T dejong (T ∗v, size_t n)

  Function 2. Implementation of 1st De Jong's mathematical function.
- static T rosenbrok (T ∗v, size_t n)

  Function 3. Implementation of the Rosenbrock mathematical function.
- static T rastrigin (T ∗v, size_t n)

  Function 4. Implementation of the Rastrigin mathematical function.
- static T griewangk (T ∗v, size_t n)

  Function 5. Implementation of the Griewangk mathematical function.
- static T sineEnvelopeSineWave (T ∗v, size_t n)

  Function 6. Implementation of the Sine Envelope Sine Wave mathematical function.
- static T stretchedVSineWave (T ∗v, size_t n)

  Function 7. Implementation of the Stretched V Sine Wave mathematical function.
- static T ackleysOne (T ∗v, size_t n)

  Function 8. Implementation of Ackley's One mathematical function.
- static T ackleysTwo (T ∗v, size_t n)

  Function 9. Implementation of Ackley's Two mathematical function.
- static T eggHolder (T ∗v, size_t n)

  Function 10. Implementation of the Egg Holder mathematical function.
- static T rana (T ∗v, size_t n)

  Function 11. Implementation of the Rana mathematical function.
- static T pathological (T ∗v, size_t n)

  Function 12. Implementation of the Pathological mathematical function.
- static T mastersCosineWave (T ∗v, size_t n)

  Function 14. Implementation of the Masters Cosine Wave mathematical function.
- static T michalewicz (T ∗v, size_t n)

  Function 13. Implementation of the Michalewicz mathematical function.
- static T quartic (T ∗v, size_t n)

  Function 15. Implementation of the Quartic mathematical function.
- static T levy (T ∗v, size_t n)

  Function 16. Implementation of the Levy mathematical function.
- static T step (T ∗v, size_t n)

  Function 17. Implementation of the Step mathematical function.
- static T alpine (T ∗v, size_t n)

Function 18. Implementation of the Alpine mathematical function.
- static mfuncPtr< T > get (unsigned int f)

  Returns a function pointer to the math function with the given id.
- static bool exec (unsigned int f, T ∗v, size_t n, T &outResult)

  Executes a specific function Executes the function with the given id and returns true on success. Otherwise returns false if id is invalid.
- static T nthroot (T x, T n)
- static T w (T x)

6.6.1 Detailed Description

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

Struct containing all static math functions. A function can be called directly by name, or indirectly using Functions↩ ::get or Functions::exec.

Template Parameters

T	Data type for function calculations

Definition at line 112 of file mfunctions.h.

6.6.2 Member Function Documentation

6.6.2.1 ackleysOne()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 331 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = (static_cast<T>(1.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2))) * sqrt(v[i]*v[i] + v[i+1]*v[i+1]);
        T b = static_cast<T>(3.0) * (cos(static_cast<T>(2.0) * v[i]) + sin(static_cast<T>(2.0) * v[i+1]));
        f += a + b;
    }

    return f;
}
```

6.6.2.2 ackleysTwo()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 355 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = static_cast<T>(20.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2) * sqrt((v[i]*v[i] + v[i+1]*v[i+1]) / static_cast<T>(2.0)));
        T b = pow(static_cast<T>(M_E), static_cast<T>(0.5) *
              (cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i+1])));
        f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
    }

    return f;
}
```

6.6.2.3 alpine()

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

Function 18. Implementation of the Alpine mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 585 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += std::abs(v[i] * sin(v[i]) + static_cast<T>(0.1)*v[i]);
    }

    return f;
}
```

6.6.2.4 dejong()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 183 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += v[i] * v[i];
    }

    return f;
}
```

6.6.2.5 eggHolder()

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

Function 10. Implementation of the Egg Holder mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 380 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = static_cast<T>(-1.0) * v[i] * sin(sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0))));
        T b = (v[i+1] + static_cast<T>(47)) * sin(sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (v[i]/static_cast<T>(2.0)))));
        f += a - b;
    }

    return f;
}
```

6.6.2.6 exec()

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

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

Parameters

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 667 of file mfunctions.h.

```
{
    auto fPtr = get(f);
    if (fPtr == nullptr) return false;

    outResult = fPtr(v, n);
    return true;
}
```

6.6.2.7 get()

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

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

Template Parameters

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 609 of file mfunctions.h.

```
{
    switch (f)
    {
        case 1:
            return Functions<T>::schwefel;
        case 2:
            return Functions<T>::dejong;
        case 3:
            return Functions<T>::rosenbrok;
        case 4:
            return Functions<T>::rastrigin;
        case 5:
            return Functions<T>::griewangk;
        case 6:
            return Functions<T>::sineEnvelopeSineWave;
        case 7:
            return Functions<T>::stretchedVSineWave;
        case 8:
            return Functions<T>::ackleysOne;
        case 9:
            return Functions<T>::ackleysTwo;
        case 10:
            return Functions<T>::eggHolder;
        case 11:
            return Functions<T>::rana;
        case 12:
            return Functions<T>::pathological;
        case 13:
            return Functions<T>::michalewicz;
        case 14:
            return Functions<T>::mastersCosineWave;
        case 15:
            return Functions<T>::quartic;
        case 16:
            return Functions<T>::levy;
        case 17:
            return Functions<T>::step;
        case 18:
            return Functions<T>::alpine;
        default:
            return nullptr;
    }
}
```

6.6.2.8 griewangk()

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

Function 5. Implementation of the Griewangk mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 252 of file mfunctions.h.

```
{
    T sum = 0.0;
    T product = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        sum += (v[i] * v[i]) / static_cast<T>(4000.0);
    }

    for (size_t i = 0; i < n; i++)
    {
        product *= cos(v[i] / sqrt(static_cast<T>(i + 1.0)));
    }

    return static_cast<T>(1.0) + sum - product;
}
```

6.6.2.9 levy()

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

Function 16. Implementation of the Levy mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 531 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = w(v[i]) - static_cast<T>(1.0);
        a *= a;
        T b = sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
        b *= b;
        T c = w(v[n - 1]) - static_cast<T>(1.0);
        c *= c;
        T d = sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
        d *= d;
        f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
    }

    T e = sin(static_cast<T>(M_PI) * w(v[0]));
    return e*e + f;
}
```

6.6.2.10 mastersCosineWave()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 476 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = pow(M_E, static_cast<T>(-1.0/8.0)*(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i+1]*v[i]));
        T b = cos(static_cast<T>(4) * sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v[i+1]));
        f += a * b;
    }

    return static_cast<T>(-1.0) * f;
}
```

6.6.2.11 michalewicz()

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

Function 13. Implementation of the Michalewicz mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 454 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += sin(v[i]) * pow(sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)), static_cast<T>(20));
    }

    return -1.0 * f;
}
```

6.6.2.12 nthroot()

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

Simple helper function that returns the nth-root

Parameters

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

Returns

The value of the nth-root of x

Definition at line 146 of file mfunctions.h.

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

6.6.2.13 pathological()

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

Function 12. Implementation of the Pathological mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 428 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = sin(sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
        a = (a*a) - static_cast<T>(0.5);
        T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
        b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
        f += static_cast<T>(0.5) + (a/b);
    }

    return f;
}
```

6.6.2.14 quartic()

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

Function 15. Implementation of the Quartic mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 500 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (i+1) * v[i] * v[i] * v[i] * v[i];
    }

    return f;
}
```

6.6.2.15 rana()

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

Function 11. Implementation of the Rana mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 404 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = v[i] * sin(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * cos(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
        T b = (v[i+1] + static_cast<T>(1.0)) * cos(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * sin(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
        f += a + b;
    }

    return f;
}
```

6.6.2.16 rastrigin()

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

Function 4. Implementation of the Rastrigin mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 230 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (v[i] * v[i]) - (static_cast<T>(10.0) * cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]));
    }

    return static_cast<T>(10.0) * static_cast<T>(n) * f;
}
```

6.6.2.17 rosenbrok()

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

Function 3. Implementation of the Rosenbrock mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 205 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = ((v[i] * v[i]) - v[i+1]);
        T b = (static_cast<T>(1.0) - v[i]);
        f += static_cast<T>(100.0) * a * a;
        f += b * b;
    }

    return f;
}
```

6.6.2.18 schwefel()

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

Function 1. Implementation of Schwefel's mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 161 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (static_cast<T>(-1.0) * v[i]) * sin(sqrt(std::abs(v[i])));
    }

    return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
}
```

6.6.2.19 sineEnvelopeSineWave()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 280 of file mfunctions.h.

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

6.6.2.20 step()

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

Function 17. Implementation of the Step mathematical function.

Parameters

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

Returns

The result of the mathematical function

Definition at line 562 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        T a = std::abs(v[i]) + static_cast<T>(0.5);
        f += a * a;
    }

    return f;
}
```

6.6.2.21 stretchedVSineWave()

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

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

Parameters

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

Returns

The result of the mathematical function

Definition at line 306 of file mfunctions.h.

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
        T b = sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0)));
        b *= b;
        f += a * b + static_cast<T>(1.0);
    }

    return f;
}
```

6.6.2.22 w()

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

Helper math function used in levy()

Definition at line 518 of file mfunctions.h.

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

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

- include/mfunctions.h

6.7 util::IniReader Class Reference

The IniReader class is a simple ∗.ini file reader and parser.

`#include <inireader.h>`

Public Member Functions

- IniReader ()

  Construct a new IniReader object.
- ∼IniReader ()

  Destroys the IniReader object.
- bool openFile (std::string filePath)

  Opens the given ini file and parses all sections/entries. The all file data is stored in memory and the file is closed.
- bool sectionExists (std::string section)

  Returns true if the given section exists in the current ini file.
- bool entryExists (std::string section, std::string entry)

  Returns true if the given section and entry key exists in the current ini file.
- std::string getEntry (std::string section, std::string entry)

  Returns the value for the entry that has the given entry key within the given section.
- template<class T >
  T getEntryAs (std::string section, std::string entry)

6.7.1 Detailed Description

The IniReader class is a simple ∗.ini file reader and parser.

– Initialize an IniReader object:

IniReader ini;

Open and parse an ∗.ini file:

ini.openFile("my_ini_file.ini");

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

Retrieve an entry from the ini file:

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

Definition at line 46 of file inireader.h.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 IniReader()

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

Construct a new IniReader object.

Definition at line 21 of file inireader.cpp.

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

6.7.2.2 ∼IniReader()

```
IniReader::∼IniReader ( )
```

Destroys the IniReader object.

Definition at line 28 of file inireader.cpp.

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

6.7.3 Member Function Documentation

6.7.3.1 entryExists()

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

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

Parameters

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().

```
{
    auto it = iniMap.find(section);
    if (it == iniMap.end()) return false;

    return it->second.find(entry) != it->second.end();
}
```

6.7.3.2 getEntry()

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

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

Parameters

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 >::testFuncThreaded().

```
{
    if (!entryExists(section, entry)) return std::string();

    return iniMap[section][entry];
}
```

6.7.3.3 getEntryAs()

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

Definition at line 57 of file inireader.h.

References getEntry().

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

```
    {
        std::stringstream ss(getEntry(section, entry));
        T retVal;
        ss >> retVal;
        return retVal;
    }
```

6.7.3.4 openFile()

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

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

Parameters

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().

```
{
    file = filePath;
    if (!parseFile())
        return false;

    return true;
}
```

6.7.3.5 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:

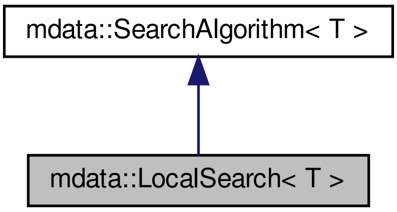
- include/inireader.h
- src/inireader.cpp

6.8 mdata::LocalSearch< T > Class Template Reference

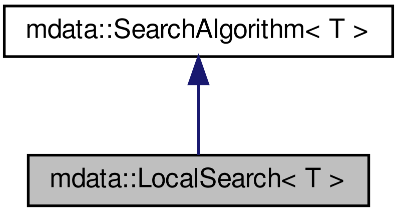
The LocalSearch class implements the Local Search algorithm, which is ran using the overridden SearchAlgorithm::run() function.

`#include <localsearch.h>`

Inheritance diagram for mdata::LocalSearch< T >:



Collaboration diagram for mdata::LocalSearch< T >:



Public Member Functions

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

  Executes Local Search with the given population and parameters.

Additional Inherited Members

6.8.1 Detailed Description

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

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

Template Parameters

T	Data type used

Definition at line 32 of file localsearch.h.

6.8.2 Member Function Documentation

6.8.2.1 run()

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

Executes Local Search with the given population and parameters.

Parameters

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

Returns

TestResult<T> Returns a TestResult struct containing the error code, fitness, and execution time

Implements mdata::SearchAlgorithm< T >.

Definition at line 49 of file localsearch.h.

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

```
        {
            // Get population size and dimensions
            const size_t popSize = pop->getPopulationSize();
            const size_t dimSize = pop->getDimensionsSize();

            // Make sure funcPtr is valid;
            if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with error code 1

            // Algorithm related variables
            bool stop = false;
            size_t pIndex = 0;

            startTimer();

            // Start recording execution time
            pop->generate(fMin, fMax);

            for (size_t sol = 0; sol < popSize; sol++)
            {
                // Populate fitness values using given math function pointer
                if (!pop->calcFitness(sol, funcPtr))
                    return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
            }

            // Get the index for the best fitness in the population
            pIndex = pop->getBestFitnessIndex();

            // Get population vector and fitness of best solution
            T* x = pop->getPopulationPtr(pIndex);
            T xFit = pop->getFitness(pIndex);

            // Create empty Y vector
            T* y = util::allocArray<T>(dimSize);
            T yFit = 0;

            // Create empty Z vector
            T* z = util::allocArray<T>(dimSize);
            T zFit = 0;

            if (x == nullptr || y == nullptr || z == nullptr)
            {
                std::cerr << "Error in Local Search: Memory allocation failed" << std::endl;
                return TestResult<T>(3, 0, 0.0);
            }

            // Keep looping until search fails to improve
            while (!stop)
            {
                stop = true;

                // Copy values from X vector into Y vector
                util::copyArray<T>(x, y, dimSize);

                // Loop through each dimension in vector
                for (size_t a = 0; a < dimSize; a++)
                {
                    // Add alpha value to y[a]
                    y[a] = x[a] + alpha;

                    // Make sure y[a] is within the function bounds
                    lockBounds(y[a], fMin, fMax);

                    // Calculate fitness for y vector
                    yFit = funcPtr(y, dimSize);

                    // Update Z[a] vector value based on the difference between Y and X
                    z[a] = x[a] - (alpha * (yFit - xFit));

                    // Make sure z[a] is within the function bounds
                    lockBounds(z[a], fMin, fMax);

                    y[a] = x[a]; // Reset y[a] to prepare for next loop
                }

                zFit = funcPtr(z, dimSize);

                // The following 'if' statement may cause extreme execution
                // times for some functions due to floating point precision:
                // if (zFit < xFit)
                //
                // The replacement 'if' statement below places a limit
                // on the minimum acknowledged improvement fitness,
                // hopefully preventing extreme run-times:
                if (xFit - zFit > MIN_IMPROVEMENT)
                {
                    // Z is an improvement on X, so keep searching
                    stop = false;

                    // Swap Z and X for next loop
                    T* tmp = x;
                    x = z;
                    xFit = zFit;
                    z = tmp;
                }
            }

            // Return best result
            return TestResult<T>(0, xFit, stopTimer());
        }
```

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

- include/localsearch.h

6.9 mdata::Population< T > Class Template Reference

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

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

Public Member Functions

- Population (size_t popSize, size_t dimensions)

  Construct a new Population object.
- ∼Population ()

  Destroy Population object.
- bool isReady ()

  Returns true if the population instance is allocated and ready to be used.
- size_t getPopulationSize ()

  Returns the size of the population.
- size_t getDimensionsSize ()

  Returns the dimensions of the population.
- T ∗ getPopulationPtr (size_t popIndex)

  Returns an array for the population with the given index.
- bool generate (T minBound, T maxBound)

  Generates new random values for this population that are within the given bounds. Resets all fitness values to zero.
- bool setFitness (size_t popIndex, T value)

Sets the fitness value for a specific population vector index.

- bool calcFitness (size_t popIndex, mfunc::mfuncPtr< T > funcPtr)

  Uses the given function pointer to update the fitness value for the population vector at the given index.
- T getFitness (size_t popIndex)

  Returns the fitness value for a specific population vector index.
- T * getFitnessPtr (size_t popIndex)

  Returns the fitness value for a specific population vector index.
- std::vector< T > getAllFitness ()

  Returns a std::vector of all current fitness values.
- T * getBestFitnessPtr ()

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

  Returns the index of the current best fitness value.
- void outputPopulation (std::ostream &outStream, const char *delim, const char *lineBreak)

  Outputs all population data to the given output stream.
- void outputFitness (std::ostream &outStream, const char *delim, const char *lineBreak)

  Outputs all fitness data to the given output stream.

6.9.1 Detailed Description

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

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

Template Parameters

T	Data type of the population.

Definition at line 30 of file population.h.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 Population()

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

Construct a new Population object.

Template Parameters

T	Data type of the population.

Parameters

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

Definition at line 27 of file population.cpp.

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

6.9.2.2 ∼Population()

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

Destroy Population object.

Template Parameters

T	Data type of the population.

Definition at line 39 of file population.cpp.

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

6.9.3 Member Function Documentation

6.9.3.1 calcFitness()

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

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

Template Parameters

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 163 of file population.cpp.

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

```
{
    if (popFitness == nullptr || popIndex >= popSize) return false;

    popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);

    return true;
}
```

6.9.3.2 generate()

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

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

Template Parameters

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 108 of file population.cpp.

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

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

    // Generate a new seed for the mersenne twister engine
    rgen = std::mt19937(rdev());

    // Set up a normal (bell-shaped) distribution for the random number generator with the correct function bounds
    std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);

    // Generate values for all vectors in popMatrix
    for (size_t s = 0; s < popSize; s++)
    {
        for (size_t d = 0; d < popDim; d++)
        {
            T rand = (T)dist(rgen);
            popMatrix[s][d] = rand;
        }
    }

    // Reset popFitness values to 0
    initArray<T>(popFitness, popSize, (T)0.0);

    return true;
}
```

6.9.3.3 getAllFitness()

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

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

Template Parameters

T	Data type of the population.

Returns

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

Definition at line 209 of file population.cpp.

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

6.9.3.4 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 233 of file population.cpp.

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

```
{
    size_t bestIndex = 0;

    for (size_t i = 1; i < popSize; i++)
    {
        if (popFitness[i] < popFitness[bestIndex])
            bestIndex = i;
    }

    return bestIndex;
}
```

6.9.3.5 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 221 of file population.cpp.

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

Referenced by mdata::BlindSearch< T >::run().

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

6.9.3.6 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 77 of file population.cpp.

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

```
{
    return popDim;
}
```

6.9.3.7 getFitness()

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

Returns the fitness value for a specific population vector index.

Template Parameters

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 180 of file population.cpp.

Referenced by mdata::LocalSearch< T >::run().

```
{
    if (popFitness == nullptr || popIndex >= popSize) return 0;

    return popFitness[popIndex];
}
```

6.9.3.8 getFitnessPtr()

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

Returns the fitness value for a specific population vector index.

Template Parameters

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 195 of file population.cpp.

```
{
    if (popFitness == nullptr || popIndex >= popSize) return 0;

    return &popFitness[popIndex];
}
```

6.9.3.9 getPopulationPtr()

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

Returns an array for the population with the given index.

Template Parameters

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 90 of file population.cpp.

Referenced by mdata::LocalSearch< T >::run().

```
{
    if (popFitness == nullptr || popIndex >= popSize) return nullptr;

    return popMatrix[popIndex];
}
```

6.9.3.10 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 65 of file population.cpp.

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

```
{
    return popSize;
}
```

6.9.3.11 isReady()

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

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

Template Parameters

T	Data type of the population.

Returns

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

Definition at line 53 of file population.cpp.

```
{
    return popMatrix != nullptr && popFitness != nullptr;
}
```

6.9.3.12 outputFitness()

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

Outputs all fitness data to the given output stream.

Template Parameters

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 281 of file population.cpp.

```
{
    if (popFitness == nullptr) return;

    for (size_t j = 0; j < popSize; j++)
    {
        outStream << popFitness[j];
            if (j < popSize - 1)
                outStream << delim;
    }

    if (lineBreak != nullptr)
        outStream << lineBreak;
}
```

6.9.3.13 outputPopulation()

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

Outputs all population data to the given output stream.

Template Parameters

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 255 of file population.cpp.

```
{
    if (popMatrix == nullptr) return;

    for (size_t j = 0; j < popSize; j++)
    {
        for (size_t k = 0; k < popDim; k++)
        {
            outStream << popMatrix[j][k];
            if (k < popDim - 1)
                outStream << delim;
        }

        outStream << lineBreak;
    }
}
```

6.9.3.14 setFitness()

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

Sets the fitness value for a specific population vector index.

Template Parameters

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 143 of file population.cpp.

```
{
    if (popFitness == nullptr || popIndex >= popSize) return false;

    popFitness[popIndex] = value;

    return true;
}
```

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

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

6.10 mfunc::RandomBounds< T > Struct Template Reference

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

`#include <experiment.h>`

Public Attributes

- T min = 0.0
- T max = 0.0

6.10.1 Detailed Description

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

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

Definition at line 35 of file experiment.h.

6.10.2 Member Data Documentation

6.10.2.1 max

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

Definition at line 38 of file experiment.h.

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

6.10.2.2 min

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

Definition at line 37 of file experiment.h.

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

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

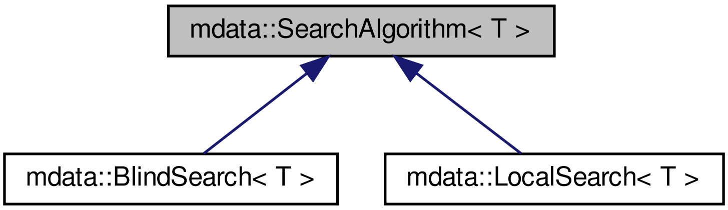
- include/experiment.h

6.11 mdata::SearchAlgorithm< T > Class Template Reference

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

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

Inheritance diagram for mdata::SearchAlgorithm< T >:



Public Member Functions

- SearchAlgorithm ()
- virtual ∼SearchAlgorithm ()=0
- virtual TestResult< T > run (mfunc::mfuncPtr< T > funcPtr, const T fMin, const T fMax, Population< T > ∗const pop, const T alpha)=0

Protected Member Functions

- void startTimer ()

  Starts the execution time timer.
- double stopTimer ()

  Returns the amount of time that has passed since startTimer() was called in miliseconds.

Protected Attributes

- double timeDiff
- high_resolution_clock::time_point timer

6.11.1 Detailed Description

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

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

Template Parameters

T	The data type used by the algorithm

Definition at line 70 of file searchalg.h.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 SearchAlgorithm()

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

Definition at line 73 of file searchalg.h.

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

6.11.2.2 ∼SearchAlgorithm()

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

Definition at line 102 of file searchalg.h.

```
00102 { }
```

6.11.3 Member Function Documentation

6.11.3.1 run()

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

Implemented in mdata::LocalSearch< T >, and mdata::BlindSearch< T >.

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

6.11.3.2 startTimer()

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

Starts the execution time timer.

Definition at line 83 of file searchalg.h.

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

```
        {
            timer = high_resolution_clock::now();
        }
```

6.11.3.3 stopTimer()

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

Returns the amount of time that has passed since startTimer() was called in miliseconds.

Definition at line 91 of file searchalg.h.

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

```
        {
            high_resolution_clock::time_point t_end = high_resolution_clock::now();
            return static_cast<double>(duration_cast<nanoseconds>(t_end - timer).count()) / 1000000.0;
        }
```

6.11.4 Member Data Documentation

6.11.4.1 timeDiff

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

Definition at line 77 of file searchalg.h.

6.11.4.2 timer

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

Definition at line 78 of file searchalg.h.

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

- include/searchalg.h

6.12 mdata::TestParameters< T > Struct Template Reference

Packs together various test experiment parameters.

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

Public Member Functions

- TestParameters ()

Public Attributes

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

6.12.1 Detailed Description

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

Packs together various test experiment parameters.

Template Parameters

T	The data type that the test algorithm and functions are using

Definition at line 29 of file testparam.h.

6.12.2 Constructor & Destructor Documentation

6.12.2.1 TestParameters()

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

Selected search algorithm for this test

Definition at line 41 of file testparam.h.

References enums::BlindSearch.

```
        {
            funcId = 1;
            alpha = 0;
            alg = enums::Algorithm::BlindSearch;
            resultsTable = nullptr;
            execTimesTable = nullptr;
            resultsCol = 0;
            execTimesCol = 0;
            resultsRow = 0;
            execTimesRow = 0;
        }
```

6.12.3 Member Data Documentation

6.12.3.1 alg

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

Pointer to the DataTable used to store the execution times

Definition at line 39 of file testparam.h.

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

6.12.3.2 alpha

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

Id for the tested math function

Definition at line 32 of file testparam.h.

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

6.12.3.3 execTimesCol

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

DataTable column index to store the fitness result

Definition at line 34 of file testparam.h.

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

6.12.3.4 execTimesRow

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

DataTable row index to store the fitness result

Definition at line 36 of file testparam.h.

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

6.12.3.5 execTimesTable

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

Pointer to the DataTable used to store the fitness results

Definition at line 38 of file testparam.h.

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

6.12.3.6 funcId

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

Definition at line 31 of file testparam.h.

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

6.12.3.7 resultsCol

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

Alpha value if needed by the search alg

Definition at line 33 of file testparam.h.

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

6.12.3.8 resultsRow

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

DataTable column index to store the execution time

Definition at line 35 of file testparam.h.

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

6.12.3.9 resultsTable

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

DataTable row index to store the execution time

Definition at line 37 of file testparam.h.

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

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

- include/testparam.h

6.13 mdata::TestResult< T > Struct Template Reference

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

Public Member Functions

- TestResult (int _err, T _fitness, double _execTime)

Public Attributes

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

6.13.1 Detailed Description

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

Definition at line 20 of file testresult.h.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 TestResult()

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

Definition at line 26 of file testresult.h.

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

6.13.3 Member Data Documentation

6.13.3.1 err

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

Definition at line 22 of file testresult.h.

6.13.3.2 execTime

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

Definition at line 24 of file testresult.h.

6.13.3.3 fitness

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

Definition at line 23 of file testresult.h.

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

- include/testresult.h

6.14 ThreadPool Class Reference

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

Public Member Functions

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

6.14.1 Detailed Description

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

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

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

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

This source file has been modified slightly by Andrew Dunn

Definition at line 42 of file threadpool.h.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 ThreadPool()

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

Definition at line 64 of file threadpool.h.

```
    :   stop(false)
{
    for(size_t i = 0;i<threads;++i)
        workers.emplace_back(
            [this]
            {
                for(;;)
                {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        this->condition.wait(lock,
                            [this]{ return this->stop || !this->tasks.empty(); });
                        if(this->stop && this->tasks.empty())
                            return;
                        task = std::move(this->tasks.front());
                        this->tasks.pop();
                    }

                    task();
                }
            }
        );
}
```

6.14.2.2 ∼ThreadPool()

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

Definition at line 117 of file threadpool.h.

References stopAndJoinAll().

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

6.14.3 Member Function Documentation

6.14.3.1 enqueue()

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

Definition at line 93 of file threadpool.h.

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

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

6.14.3.2 stopAndJoinAll()

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

Definition at line 122 of file threadpool.h.

Referenced by mfunc::Experiment< T >::testAllFunc(), and ∼ThreadPool().

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

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

- include/threadpool.h

Chapter 7

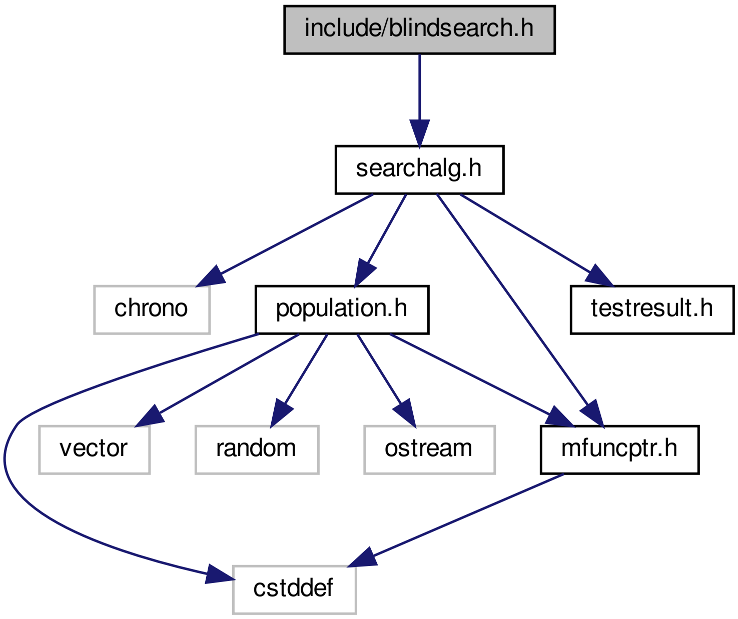
File Documentation

7.1 include/blindsearch.h File Reference

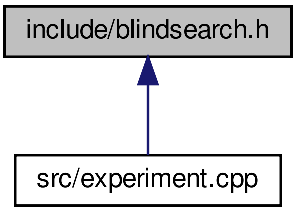
Implements the BlindSearch class, which inherits SearchAlgorithm. BlindSearch::run executes the blind search algorithm on a given population.

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

Include dependency graph for blindsearch.h:



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



Classes

- class mdata::BlindSearch< T >

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

Namespaces

- mdata

7.1.1 Detailed Description

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

Author

Andrew Dunn (`Andrew.Dunn@cwu.edu`)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file blindsearch.h.

7.2 blindsearch.h

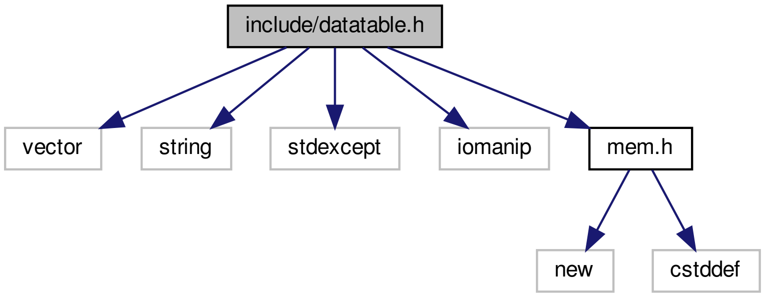
```
#ifndef __BLINDSEARCH_H
#define __BLINDSEARCH_H

#include "searchalg.h"

namespace mdata
{
    template<class T>
    class BlindSearch : public SearchAlgorithm<T>
    {
        // Declaration needed due to template base class
        using SearchAlgorithm<T>::startTimer;
        using SearchAlgorithm<T>::stopTimer;

    public:
        virtual TestResult<T> run(mfunc::mfuncPtr<T> funcPtr, const T fMin, const T fMax, Population<T>* const pop, const T alpha)
        {
            // Get population size and dimensions
            size_t popSize = pop->getPopulationSize();
            size_t dimSize = pop->getDimensionsSize();

            // Make sure funcPtr is valid;
            if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with error code 1

            // Start recording execution time
            startTimer();

            // Generate values for population vector matrix
            pop->generate(fMin, fMax);

            // For each population vector, calculate the fitness using the funcPtr
            for (size_t sol = 0; sol < popSize; sol++)
            {
                // Populate fitness values using given math function pointer
                if (!pop->calcFitness(sol, funcPtr))
                    return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
            }

            // Return best fitness value in population
            return TestResult<T>(0, *pop->getBestFitnessPtr(), stopTimer());
        }
    };
}

#endif

// ==========================
// End of blindsearch.h
// ==========================
```

7.3 include/datatable.h File Reference

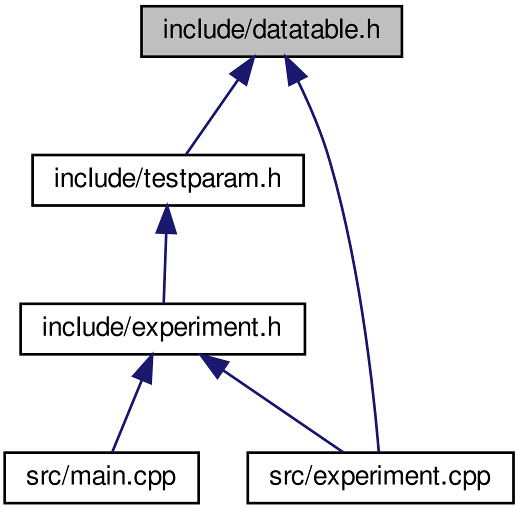
Header file for the DataTable class, which represents a spreadsheet/table of values that can easily be exported to a *.csv file.

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

Include dependency graph for datatable.h:



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



Classes

- class mdata::DataTable< T >

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

Namespaces

- mdata

7.3.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file datatable.h.

7.4 datatable.h

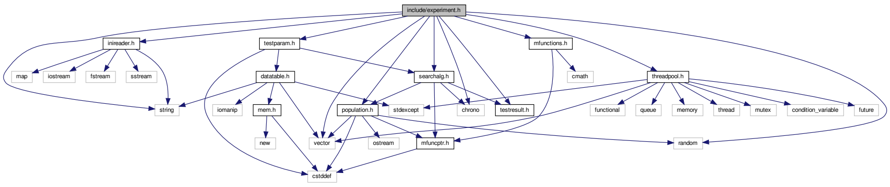
```

#ifndef __DATATABLE_H
#define __DATATABLE_H

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

namespace mdata
{
    template <class T>
    class DataTable
    {
    public:
        DataTable(size_t _rows, size_t _cols) : rows(_rows), cols(_cols), dataMatrix(nullptr)
        {
            if (rows == 0)
                throw std::length_error("Table rows must be greater than 0.");
            else if (cols == 0)
                throw std::length_error("Table columns must be greater than 0.");

            dataMatrix = util::allocMatrix<T>(rows, cols);
            if (dataMatrix == nullptr)
                throw std::bad_alloc();

            colLabels.resize(_cols, std::string());
        }

        ~DataTable()
        {
            util::releaseMatrix(dataMatrix, rows);
        }

        std::string getColLabel(size_t colIndex)
        {
            if (colIndex >= colLabels.size())
                throw std::out_of_range("Column index out of range");

            return colLabels[colIndex];
        }

        void setColLabel(size_t colIndex, std::string newLabel)
        {
             if (colIndex >= colLabels.size())
                throw std::out_of_range("Column index out of range");

            colLabels[colIndex] = newLabel;
        }

        T getEntry(size_t row, size_t col)
        {
            if (dataMatrix == nullptr)
                throw std::runtime_error("Data matrix not allocated");
            if (row >= rows)
                throw std::out_of_range("Table row out of range");
            else if (col >= cols)
                throw std::out_of_range("Table column out of range");

            return dataMatrix[row][col];
        }

        void setEntry(size_t row, size_t col, T val)
        {
            if (dataMatrix == nullptr)
                throw std::runtime_error("Data matrix not allocated");
            if (row >= rows)
                throw std::out_of_range("Table row out of range");
            else if (col >= cols)
                throw std::out_of_range("Table column out of range");

            dataMatrix[row][col] = val;
        }

        bool exportCSV(const char* filePath)
        {
            if (dataMatrix == nullptr) return false;

            using namespace std;
            ofstream outFile;
            outFile.open(filePath, ofstream::out | ofstream::trunc);
            if (!outFile.good()) return false;

            // Print column labels
            for (unsigned int c = 0; c < cols; c++)
            {
                outFile << colLabels[c];
                if (c < cols - 1) outFile << ",";
            }

            outFile << endl;

            // Print data rows
            for (unsigned int r = 0; r < rows; r++)
            {
                for (unsigned int c = 0; c < cols; c++)
                {
                    outFile << std::setprecision(8) << dataMatrix[r][c];
                    if (c < cols - 1) outFile << ",";
                }
                outFile << endl;
            }

            outFile.close();
            return true;
        }
    private:
        size_t rows;
        size_t cols;
        std::vector<std::string> colLabels;
        T** dataMatrix;
    };
} // mdata

#endif

// =========================
// End of datatable.h
// =========================
```

7.5 include/experiment.h File Reference

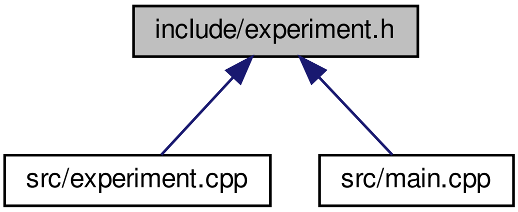
Header file for the Experiment class. Contains the basic logic and functions to run the cs471 project experiment.

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

Include dependency graph for experiment.h:



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



Classes

- struct mfunc::RandomBounds< T >

  Simple struct for storing the minimum and maximum input vector bounds for a function.
- class mfunc::Experiment< T >

  Contains classes for running the CS471 project experiment.

Namespaces

- mfunc

7.5.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file experiment.h.

7.6 experiment.h

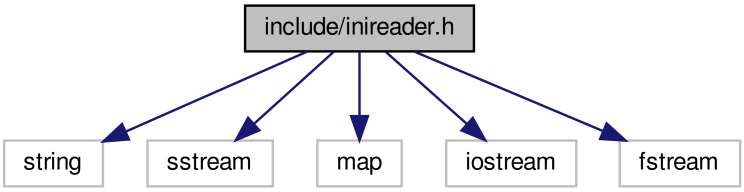
```

#ifndef __EXPERIMENT_H
#define __EXPERIMENT_H

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

namespace mfunc
{
    template<class T>
    struct RandomBounds
    {
        T min = 0.0;
        T max = 0.0;
    };

    template<class T>
    class Experiment
    {
    public:
        Experiment();
        ~Experiment();
        bool init(const char* paramFile);
        int testAllFunc();
        int testFuncThreaded(mdata::TestParameters<T> tParams);
    private:
        std::mutex popPoolMutex;
        util::IniReader iniParams;
        std::vector<mdata::Population<T>*> populationsPool;
        std::string resultsFile;
        std::string execTimesFile;
        RandomBounds<T>* vBounds;
        ThreadPool* tPool;
        size_t iterations;
        T alpha;
        enums::Algorithm testAlg;

        mdata::Population<T>* popPoolRemove();
        void popPoolAdd(mdata::Population<T>* popPtr);

        bool parseFuncBounds();

        bool allocatePopulationPool(size_t count, size_t popSize, size_t dimensions);
        void releasePopulationPool();

        bool allocateVBounds();
        void releaseVBounds();

        bool allocateThreadPool(size_t numThreads);
        void releaseThreadPool();
    };
} // mfunc

#endif

// ========================
// End of experiment.h
// ========================
```

7.7 include/inireader.h File Reference

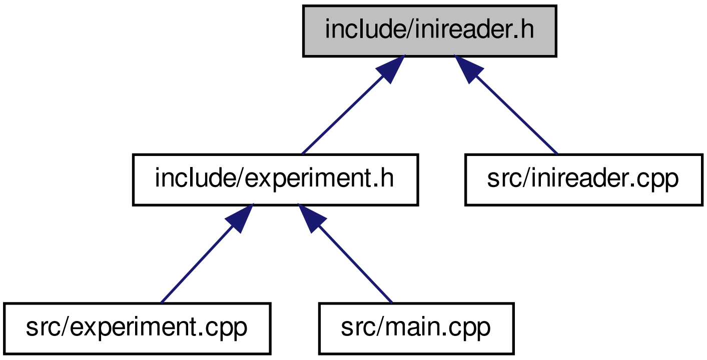
Header file for the IniReader class, which can open and parse simple ∗.ini files.

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

Include dependency graph for inireader.h:



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



Classes

- class util::IniReader

  The IniReader class is a simple ∗.ini file reader and parser.

Namespaces

- util

7.7.1 Detailed Description

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

Author

Andrew Dunn (`Andrew.Dunn@cwu.edu`)

Version

0.1

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file inireader.h.

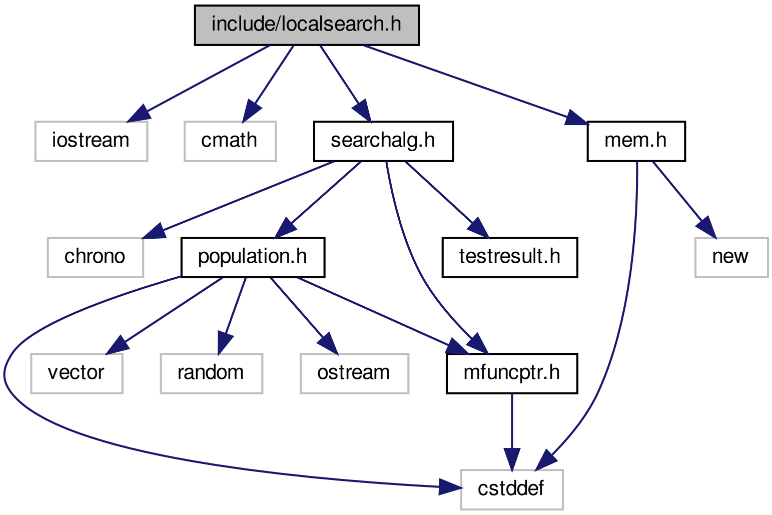
7.8 inireader.h

```

#ifndef __INIREADER_H
#define __INIREADER_H

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

namespace util
{
    class IniReader
    {
    public:
        IniReader();
        ~IniReader();
        bool openFile(std::string filePath);
        bool sectionExists(std::string section);
        bool entryExists(std::string section, std::string entry);
        std::string getEntry(std::string section, std::string entry);

        template <class T>
        T getEntryAs(std::string section, std::string entry)
        {
            std::stringstream ss(getEntry(section, entry));
            T retVal;
            ss >> retVal;
            return retVal;
        }
    private:
        std::string file;
        std::map<std::string, std::map<std::string, std::string>> iniMap;
        bool parseFile();
        void parseEntry(const std::string& sectionName, const std::string& entry);
    };
}

#endif

// ========================
// End of inireader.h
// ========================
```

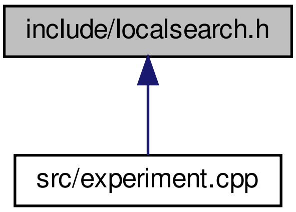
7.9 include/localsearch.h File Reference

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

Include dependency graph for localsearch.h:



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



Classes

- class mdata::LocalSearch< T >

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

Namespaces

- mdata

Macros

- #define DEC_PRECISION 12

7.9.1 Detailed Description

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file localsearch.h.

7.9.2 Macro Definition Documentation

7.9.2.1 DEC_PRECISION

```
#define DEC_PRECISION 12
```

Definition at line 21 of file localsearch.h.

7.10 localsearch.h

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

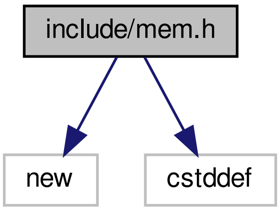
```
            // Get population size and dimensions
            const size_t popSize = pop->getPopulationSize();
            const size_t dimSize = pop->getDimensionsSize();

            // Make sure funcPtr is valid;
            if (funcPtr == nullptr) return TestResult<T>(1, 0, 0.0); // Invalid function id, return with error code 1

            // Algorithm related variables
            bool stop = false;
            size_t pIndex = 0;

            startTimer();

            // Start recording execution time
            pop->generate(fMin, fMax);

            for (size_t sol = 0; sol < popSize; sol++)
            {
                // Populate fitness values using given math function pointer
                if (!pop->calcFitness(sol, funcPtr))
                    return TestResult<T>(2, 0, 0.0); // Invalid fitness index, return with error code 2
            }

            // Get the index for the best fitness in the population
            pIndex = pop->getBestFitnessIndex();

            // Get population vector and fitness of best solution
            T* x = pop->getPopulationPtr(pIndex);
            T xFit = pop->getFitness(pIndex);

            // Create empty Y vector
            T* y = util::allocArray<T>(dimSize);
            T yFit = 0;

            // Create empty Z vector
            T* z = util::allocArray<T>(dimSize);
            T zFit = 0;

            if (x == nullptr || y == nullptr || z == nullptr)
            {
                std::cerr << "Error in Local Search: Memory allocation failed" << std::endl;
                return TestResult<T>(3, 0, 0.0);
            }

            // Keep looping until search fails to improve
            while (!stop)
            {
                stop = true;

                // Copy values from X vector into Y vector
                util::copyArray<T>(x, y, dimSize);

                // Loop through each dimension in vector
                for (size_t a = 0; a < dimSize; a++)
                {
                    // Add alpha value to y[a]
                    y[a] = x[a] + alpha;

                    // Make sure y[a] is within the function bounds
                    lockBounds(y[a], fMin, fMax);

                    // Calculate fitness for y vector
                    yFit = funcPtr(y, dimSize);

                    // Update Z[a] vector value based on the difference between Y and X
                    z[a] = x[a] - (alpha * (yFit - xFit));

                    // Make sure z[a] is within the function bounds
                    lockBounds(z[a], fMin, fMax);

                    y[a] = x[a]; // Reset y[a] to prepare for next loop
                }

                zFit = funcPtr(z, dimSize);

                // The following 'if' statement may cause extreme execution
                // times for some functions due to floating point precision:
                // if (zFit < xFit)
                //
                // The replacement 'if' statement below places a limit
                // on the minimum acknowledged improvement fitness,
                // hopefully preventing extreme run-times:
                if (xFit - zFit > MIN_IMPROVEMENT)
                {
                    // Z is an improvement on X, so keep searching
                    stop = false;

                    // Swap Z and X for next loop
                    T* tmp = x;
                    x = z;
                    xFit = zFit;
                    z = tmp;
                }
            }

            // Return best result
            return TestResult<T>(0, xFit, stopTimer());
        }
    private:
        void lockBounds(T& val, const T& min, const T& max)
        {
            if (val < min) val = min;
            else if (val > max) val = max;
        }
    };
}

#endif

// ==========================
// End of localsearch.h
// ==========================
```

7.11 include/mem.h File Reference

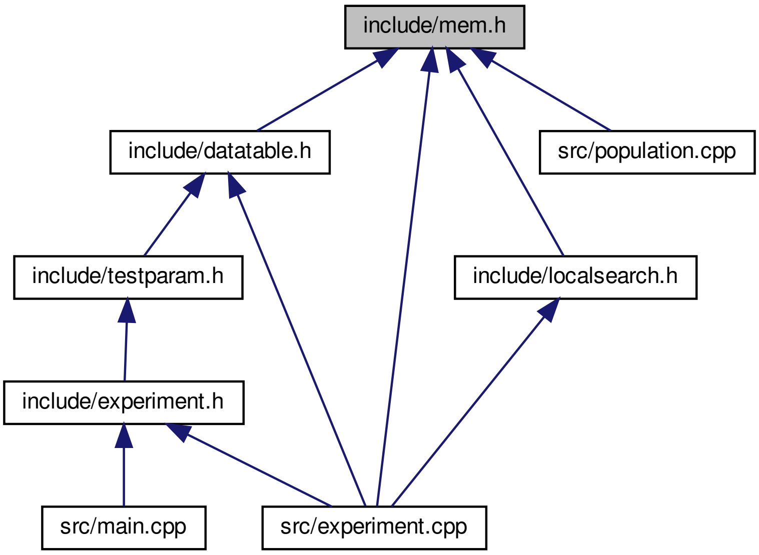
Header file for various memory utility functions.

```
#include <new>
#include <cstddef>
```

Include dependency graph for mem.h:



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



Namespaces

- util

Functions

- template<class T = double>
  void util::initArray (T ∗a, size_t size, T val)

  Initializes an array with some set value.
- template<class T = double>
  void util::initMatrix (T ∗∗m, size_t rows, size_t cols, T val)

  Initializes a matrix with a set value for each entry.
- template<class T = double>
  void util::releaseArray (T ∗&a)

  Releases an allocated array's memory and sets the pointer to nullptr.
- template<class T = double>
  void util::releaseMatrix (T ∗∗&m, size_t rows)

  Releases an allocated matrix's memory and sets the pointer to nullptr.
- template<class T = double>
  T ∗ util::allocArray (size_t size)

  Allocates a new array of the given data type.
- template<class T = double>
  T ∗∗ util::allocMatrix (size_t rows, size_t cols)

  Allocates a new matrix of the given data type.
- template<class T = double>
  void util::copyArray (T ∗src, T ∗dest, size_t size)

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

7.11.1 Detailed Description

Header file for various memory utility functions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-02

Copyright

Copyright (c) 2019

Definition in file mem.h.

7.12 mem.h

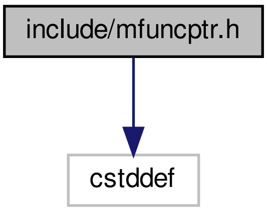
```

#ifndef __MEM_H
#define __MEM_H

#include <new> // std::nothrow
#include <cstddef> // size_t definition

namespace util
{
    template <class T = double>
    inline void initArray(T* a, size_t size, T val)
    {
        if (a == nullptr) return;

        for (size_t i = 0; i < size; i++)
        {
            a[i] = val;
        }
    }

    template <class T = double>
    inline void initMatrix(T** m, size_t rows, size_t cols, T val)
    {
        if (m == nullptr) return;

        for (size_t i = 0; i < rows; i++)
        {
            initArray(m[i], cols, val);
        }
    }

    template <class T = double>
    void releaseArray(T*& a)
    {
        if (a == nullptr) return;

        delete[] a;
        a = nullptr;
    }

    template <class T = double>
    void releaseMatrix(T**& m, size_t rows)
    {
        if (m == nullptr) return;

        for (size_t i = 0; i < rows; i++)
        {
            if (m[i] != nullptr)
            {
                // Release each row
                releaseArray<T>(m[i]);
            }
        }

        // Release columns
        delete[] m;
        m = nullptr;
    }

    template <class T = double>
    inline T* allocArray(size_t size)
    {
        return new(std::nothrow) T[size];
    }

    template <class T = double>
    inline T** allocMatrix(size_t rows, size_t cols)
    {
        T** m = (T**)allocArray<T*>(rows);
        if (m == nullptr) return nullptr;

        for (size_t i = 0; i < rows; i++)
        {
            m[i] = allocArray<T>(cols);
            if (m[i] == nullptr)
            {
                releaseMatrix<T>(m, rows);
                return nullptr;
            }
        }

        return m;
    }

    template <class T = double>
    inline void copyArray(T* src, T* dest, size_t size)
    {
        for (size_t i = 0; i < size; i++)
            dest[i] = src[i];
    }
}

#endif

// ========================
// End of mem.h
// ========================
```

7.13 include/mfuncptr.h File Reference

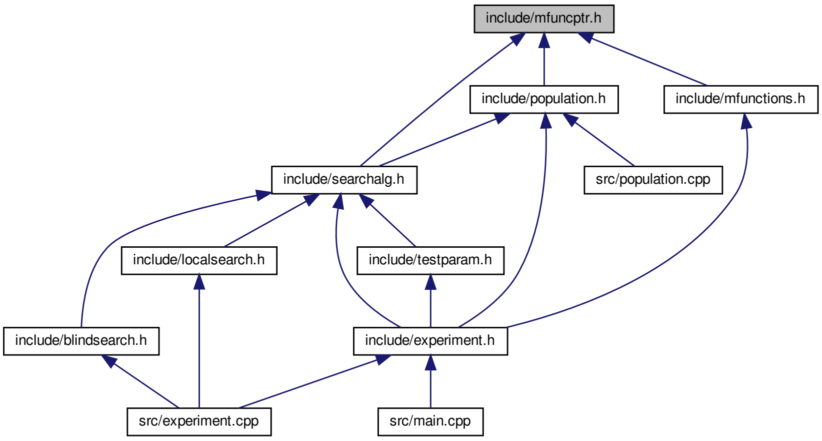
Contains the type definition for mfuncPtr, a templated function pointer to one of the math functions in mfunctions.h.

```
#include <cstddef>
```

Include dependency graph for mfuncptr.h:



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



Namespaces

- mfunc

Typedefs

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

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

7.13.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file mfuncptr.h.

7.14 mfuncptr.h

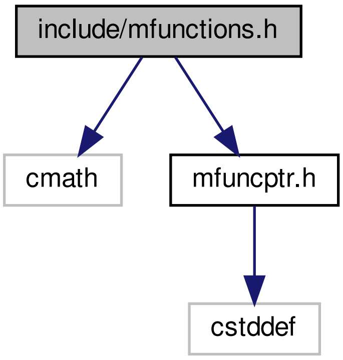
```

#ifndef __MFUNCPTR_H
#define __MFUNCPTR_H

#include <cstddef> // size_t definition

namespace mfunc
{
    template <class T>
    using mfuncPtr = T (*)(T*, size_t);
}

#endif

// =========================
// End of mfuncptr.h
// =========================
```

7.15 include/mfunctions.h File Reference

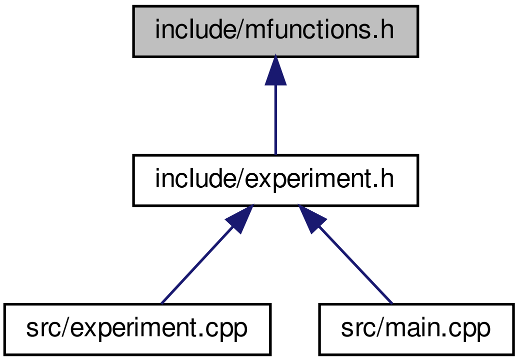
Contains various math function definitions.

```
#include <cmath>
#include "mfuncptr.h"
```

Include dependency graph for mfunctions.h:



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



Classes

- struct mfunc::FunctionDesc

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

- struct mfunc::Functions< T >

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

Namespaces

- mfunc

Macros

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

Variables

- constexpr const unsigned int mfunc::NUM_FUNCTIONS = 18

7.15.1 Detailed Description

Contains various math function definitions.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-03-29

Copyright

Copyright (c) 2019

Definition in file mfunctions.h.

7.15.2 Macro Definition Documentation

7.15.2.1 _ackleysOneDesc

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

Definition at line 27 of file mfunctions.h.

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

7.15.2.2 _ackleysTwoDesc

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

Definition at line 28 of file mfunctions.h.

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

7.15.2.3 _alpineDesc

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

Definition at line 37 of file mfunctions.h.

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

7.15.2.4 _dejongDesc

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

Definition at line 21 of file mfunctions.h.

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

7.15.2.5 _eggHolderDesc

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

Definition at line 29 of file mfunctions.h.

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

7.15.2.6 _griewangkDesc

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

Definition at line 24 of file mfunctions.h.

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

7.15.2.7 _levyDesc

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

Definition at line 35 of file mfunctions.h.

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

7.15.2.8 _mastersCosineWaveDesc

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

Definition at line 33 of file mfunctions.h.

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

7.15.2.9 _michalewiczDesc

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

Definition at line 32 of file mfunctions.h.

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

7.15.2.10 _pathologicalDesc

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

Definition at line 31 of file mfunctions.h.

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

7.15.2.11 _quarticDesc

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

Definition at line 34 of file mfunctions.h.

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

7.15.2.12 _ranaDesc

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

Definition at line 30 of file mfunctions.h.

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

7.15.2.13 _rastriginDesc

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

Definition at line 23 of file mfunctions.h.

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

7.15.2.14 _rosenbrokDesc

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

Definition at line 22 of file mfunctions.h.

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

7.15.2.15 _schwefelDesc

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

Definition at line 20 of file mfunctions.h.

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

7.15.2.16 _sineEnvelopeSineWaveDesc

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

Definition at line 25 of file mfunctions.h.

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

7.15.2.17 _stepDesc

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

Definition at line 36 of file mfunctions.h.

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

7.15.2.18 _stretchedVSineWaveDesc

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

Definition at line 26 of file mfunctions.h.

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

7.15.2.19 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

Definition at line 15 of file mfunctions.h.

7.16 mfunctions.h

```

#ifndef __MFUNCIONS_H
#define __MFUNCIONS_H

#define _USE_MATH_DEFINES

#include <cmath>
#include "mfuncptr.h"

#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"

namespace mfunc
{
    constexpr const unsigned int NUM_FUNCTIONS = 18;

    struct FunctionDesc
    {
        static const char* get(unsigned int f)
        {
            switch (f)
            {
                case 1:
                    return _schwefelDesc;
                case 2:
                    return _dejongDesc;
                case 3:
                    return _rosenbrokDesc;
                case 4:
                    return _rastriginDesc;
                case 5:
                    return _griewangkDesc;
                case 6:
                    return _sineEnvelopeSineWaveDesc;
                case 7:
                    return _stretchedVSineWaveDesc;
                case 8:
                    return _ackleysOneDesc;
                case 9:
                    return _ackleysTwoDesc;
                case 10:
                    return _eggHolderDesc;
                case 11:
                    return _ranaDesc;
                case 12:
                    return _pathologicalDesc;
                case 13:
                    return _michalewiczDesc;
                case 14:
                    return _mastersCosineWaveDesc;
                case 15:
                    return _quarticDesc;
                case 16:
                    return _levyDesc;
                case 17:
                    return _stepDesc;
                case 18:
                    return _alpineDesc;
                default:
                    return NULL;
            }
        }
    };

    template <class T>
    struct Functions
    {
        static T schwefel(T* v, size_t n);
        static T dejong(T* v, size_t n);
        static T rosenbrok(T* v, size_t n);
        static T rastrigin(T* v, size_t n);
        static T griewangk(T* v, size_t n);
        static T sineEnvelopeSineWave(T* v, size_t n);
        static T stretchedVSineWave(T* v, size_t n);
        static T ackleysOne(T* v, size_t n);
        static T ackleysTwo(T* v, size_t n);
        static T eggHolder(T* v, size_t n);
        static T rana(T* v, size_t n);
        static T pathological(T* v, size_t n);
        static T mastersCosineWave(T* v, size_t n);
        static T michalewicz(T* v, size_t n);
        static T quartic(T* v, size_t n);
        static T levy(T* v, size_t n);
        static T step(T* v, size_t n);
        static T alpine(T* v, size_t n);
        static mfuncPtr<T> get(unsigned int f);
        static bool exec(unsigned int f, T* v, size_t n, T& outResult);
        static T nthroot(T x, T n);
        static T w(T x);
    };
}

template <class T>
T mfunc::Functions<T>::nthroot(T x, T n)
{
    return pow(x, static_cast<T>(1.0) / n);
}

// ================================================

template <class T>
T mfunc::Functions<T>::schwefel(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (static_cast<T>(-1.0) * v[i]) * sin(sqrt(std::abs(v[i])));
    }

    return (static_cast<T>(418.9829) * static_cast<T>(n)) - f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::dejong(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += v[i] * v[i];
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::rosenbrok(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = ((v[i] * v[i]) - v[i+1]);
        T b = (static_cast<T>(1.0) - v[i]);
        f += static_cast<T>(100.0) * a * a;
        f += b * b;
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::rastrigin(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (v[i] * v[i]) - (static_cast<T>(10.0) * cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]));
    }

    return static_cast<T>(10.0) * static_cast<T>(n) * f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::griewangk(T* v, size_t n)
{
    T sum = 0.0;
    T product = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        sum += (v[i] * v[i]) / static_cast<T>(4000.0);
    }

    for (size_t i = 0; i < n; i++)
    {
        product *= cos(v[i] / sqrt(static_cast<T>(i + 1.0)));
    }

    return static_cast<T>(1.0) + sum - product;
}

// ================================================

template <class T>
T mfunc::Functions<T>::sineEnvelopeSineWave(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = sin(v[i]*v[i] + v[i+1]*v[i+1] - static_cast<T>(0.5));
        a *= a;
        T b = (static_cast<T>(1.0) + static_cast<T>(0.001)*(v[i]*v[i] + v[i+1]*v[i+1]));
        b *= b;
        f += static_cast<T>(0.5) + (a / b);
    }

    return static_cast<T>(-1.0) * f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::stretchedVSineWave(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(4.0));
        T b = sin(static_cast<T>(50.0) * nthroot(v[i]*v[i] + v[i+1]*v[i+1], static_cast<T>(10.0)));
        b *= b;
        f += a * b + static_cast<T>(1.0);
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::ackleysOne(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = (static_cast<T>(1.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2))) * sqrt(v[i]*v[i] + v[i+1]*v[i+1]);
        T b = static_cast<T>(3.0) * (cos(static_cast<T>(2.0) * v[i]) + sin(static_cast<T>(2.0) * v[i+1]));
        f += a + b;
    }

    return f;
}

// ================================================

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

```
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = static_cast<T>(20.0) / pow(static_cast<T>(M_E), static_cast<T>(0.2) * sqrt((v[i]*v[i] + v[i+1]*v[i+1]) / static_cast<T>(2.0)));
        T b = pow(static_cast<T>(M_E), static_cast<T>(0.5) *
            (cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i]) + cos(static_cast<T>(2.0) * static_cast<T>(M_PI) * v[i+1])));
        f += static_cast<T>(20.0) + static_cast<T>(M_E) - a - b;
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::eggHolder(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = static_cast<T>(-1.0) * v[i] * sin(sqrt(std::abs(v[i] - v[i+1] - static_cast<T>(47.0))));
        T b = (v[i+1] + static_cast<T>(47)) * sin(sqrt(std::abs(v[i+1] + static_cast<T>(47.0) + (v[i]/static_cast<T>(2.0)))));
        f += a - b;
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::rana(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = v[i] * sin(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * cos(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
        T b = (v[i+1] + static_cast<T>(1.0)) * cos(sqrt(std::abs(v[i+1] - v[i] + static_cast<T>(1.0)))) * sin(sqrt(std::abs(v[i+1] + v[i] + static_cast<T>(1.0))));
        f += a + b;
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::pathological(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = sin(sqrt(static_cast<T>(100.0)*v[i]*v[i] + v[i+1]*v[i+1]));
        a = (a*a) - static_cast<T>(0.5);
        T b = (v[i]*v[i] - static_cast<T>(2)*v[i]*v[i+1] + v[i+1]*v[i+1]);
        b = static_cast<T>(1.0) + static_cast<T>(0.001) * b*b;
        f += static_cast<T>(0.5) + (a/b);
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::michalewicz(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += sin(v[i]) * pow(sin(((i+1) * v[i] * v[i]) / static_cast<T>(M_PI)), static_cast<T>(20));
    }

    return -1.0 * f;
}

```

```
// ================================================

template <class T>
T mfunc::Functions<T>::mastersCosineWave(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = pow(M_E, static_cast<T>(-1.0/8.0)*(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i+1]*v[i]));
        T b = cos(static_cast<T>(4) * sqrt(v[i]*v[i] + v[i+1]*v[i+1] + static_cast<T>(0.5)*v[i]*v[i+1]));
        f += a * b;
    }

    return static_cast<T>(-1.0) * f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::quartic(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += (i+1) * v[i] * v[i] * v[i] * v[i];
    }

    return f;
}

// ================================================

template <class T>
T  mfunc::Functions<T>::w(T x)
{
    return static_cast<T>(1.0) + (x - static_cast<T>(1.0)) / static_cast<T>(4.0);
}

template <class T>
T mfunc::Functions<T>::levy(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n - 1; i++)
    {
        T a = w(v[i]) - static_cast<T>(1.0);
        a *= a;
        T b = sin(static_cast<T>(M_PI) * w(v[i]) + static_cast<T>(1.0));
        b *= b;
        T c = w(v[n - 1]) - static_cast<T>(1.0);
        c *= c;
        T d = sin(static_cast<T>(2.0) * static_cast<T>(M_PI) * w(v[n - 1]));
        d *= d;
        f += a * (static_cast<T>(1.0) + static_cast<T>(10.0) * b) + c * (static_cast<T>(1.0) + d);
    }

    T e = sin(static_cast<T>(M_PI) * w(v[0]));
    return e*e + f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::step(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        T a = std::abs(v[i]) + static_cast<T>(0.5);
        f += a * a;
    }

    return f;
}

// ================================================

template <class T>
T mfunc::Functions<T>::alpine(T* v, size_t n)
{
    T f = 0.0;

    for (size_t i = 0; i < n; i++)
    {
        f += std::abs(v[i] * sin(v[i]) + static_cast<T>(0.1)*v[i]);
    }

    return f;
}

// ================================================

template <class T>
mfunc::mfuncPtr<T> mfunc::Functions<T>::get(unsigned int f)
{
    switch (f)
    {
        case 1:
            return Functions<T>::schwefel;
        case 2:
            return Functions<T>::dejong;
        case 3:
            return Functions<T>::rosenbrok;
        case 4:
            return Functions<T>::rastrigin;
        case 5:
            return Functions<T>::griewangk;
        case 6:
            return Functions<T>::sineEnvelopeSineWave;
        case 7:
            return Functions<T>::stretchedVSineWave;
        case 8:
            return Functions<T>::ackleysOne;
        case 9:
            return Functions<T>::ackleysTwo;
        case 10:
            return Functions<T>::eggHolder;
        case 11:
            return Functions<T>::rana;
        case 12:
            return Functions<T>::pathological;
        case 13:
            return Functions<T>::michalewicz;
        case 14:
            return Functions<T>::mastersCosineWave;
        case 15:
            return Functions<T>::quartic;
        case 16:
            return Functions<T>::levy;
        case 17:
            return Functions<T>::step;
        case 18:
            return Functions<T>::alpine;
        default:
            return nullptr;
    }
}

// ================================================

template <class T>
bool mfunc::Functions<T>::exec(unsigned int f, T* v, size_t n, T& outResult)
{
    auto fPtr = get(f);
    if (fPtr == nullptr) return false;

    outResult = fPtr(v, n);
    return true;
}

#endif

// =========================
// End of mfunctions.h
// =========================
```

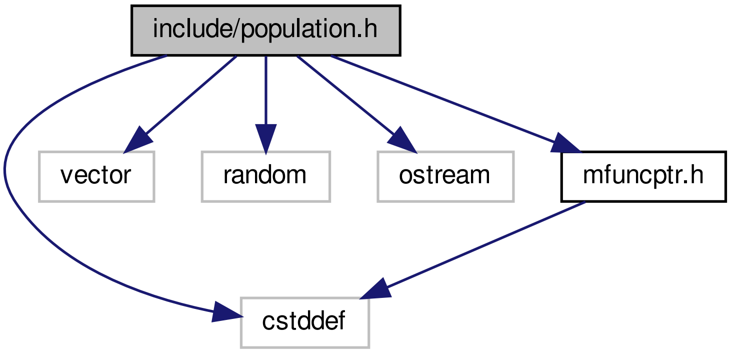
7.17 include/population.h File Reference

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

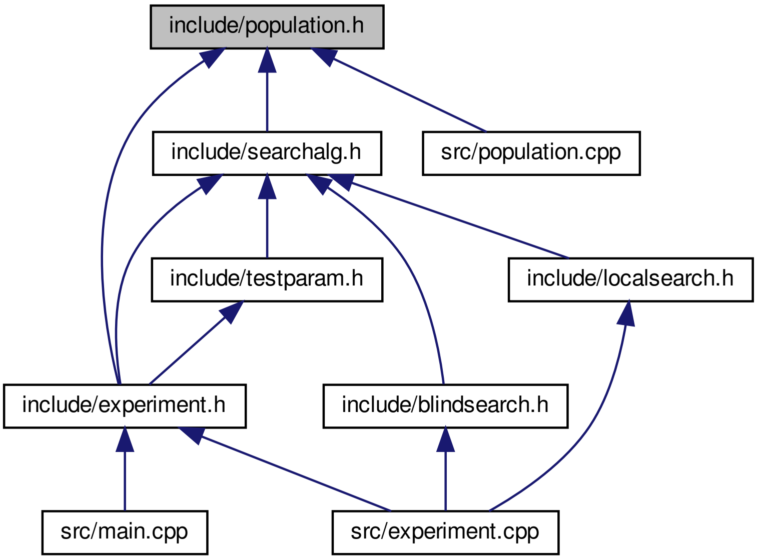
```
#include <cstddef>
#include <vector>
```

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

Include dependency graph for population.h:



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



Classes

- class mdata::Population< T >

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

Namespaces

- mdata

7.17.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-04

Copyright

Copyright (c) 2019

Definition in file population.h.

7.18 population.h

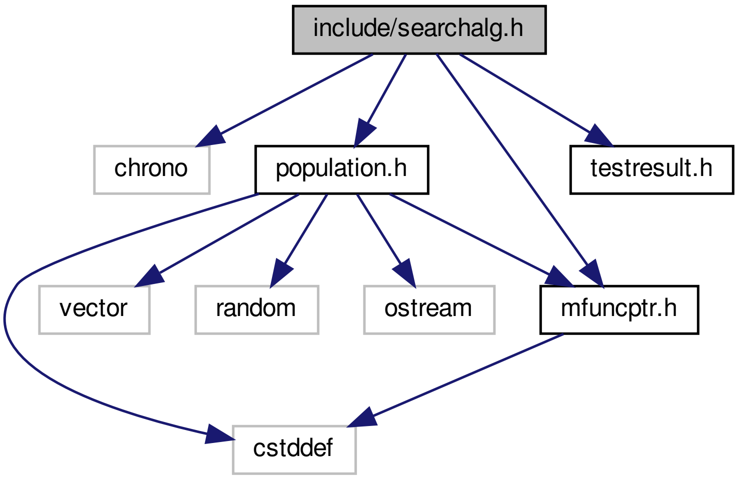
```

#ifndef __POPULATION_H
#define __POPULATION_H

#include <cstddef> // size_t definition
#include <vector>
#include <random>
#include <ostream>
#include "mfuncptr.h"

namespace mdata
{
    template<class T>
    class Population
    {
    public:
        Population(size_t popSize, size_t dimensions);
        ~Population();

        bool isReady();
        size_t getPopulationSize();
        size_t getDimensionsSize();
        T* getPopulationPtr(size_t popIndex);

        bool generate(T minBound, T maxBound);
        bool setFitness(size_t popIndex, T value);
        bool calcFitness(size_t popIndex, mfunc::mfuncPtr<T> funcPtr);

        T getFitness(size_t popIndex);
        T* getFitnessPtr(size_t popIndex);
        std::vector<T> getAllFitness();
        T* getBestFitnessPtr();
        size_t getBestFitnessIndex();

        void outputPopulation(std::ostream& outStream, const char* delim, const char* lineBreak);
        void outputFitness(std::ostream& outStream, const char* delim, const char* lineBreak);
    private:
        const size_t popSize;
        const size_t popDim;
        T** popMatrix;
        T* popFitness;
        std::random_device rdev;
        std::mt19937 rgen;
        bool allocPopMatrix();
        void releasePopMatrix();

        bool allocPopFitness();
        void releasePopFitness();
    };
}

#endif

// =========================
// End of population.h
// =========================
```

7.19 include/searchalg.h File Reference

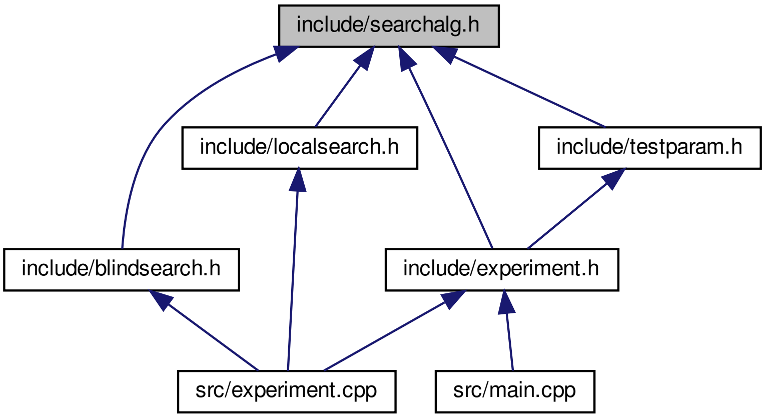
Defines the SearchAlgorithm class, Algorithm enum, and AlgorithmNames struct. The SearchAlgorithm class serves as a base class for implemented search algorithms.

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

Include dependency graph for searchalg.h:



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



Classes

- struct enums::AlgorithmNames

  Struct that contains constant string names for the different search algorithms.
- class mdata::SearchAlgorithm< T >

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

Namespaces

- enums
- mdata

Enumerations

- enum enums::Algorithm { enums::BlindSearch = 0, enums::LocalSearch = 1, enums::Count = 2 }

  Enum of different available search algorithms.

7.19.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file searchalg.h.

7.20 searchalg.h

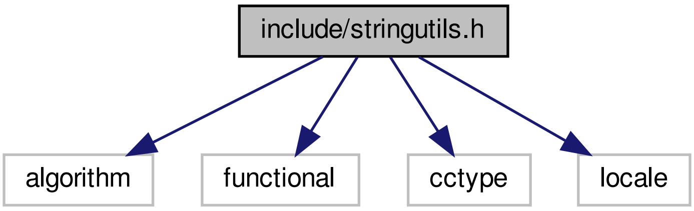
```

#ifndef __SEARCHALG_H
#define __SEARCHALG_H

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

using namespace std::chrono;

namespace enums
{
    enum Algorithm
    {
        BlindSearch = 0,
        LocalSearch = 1,
        Count = 2
    };

    struct AlgorithmNames
    {
        static constexpr const char* BLIND_SEARCH = "Blind Search";
        static constexpr const char* LOCAL_SEARCH = "Local Search";

        static const char* get(Algorithm alg)
        {
            switch (alg)
            {
                case Algorithm::BlindSearch:
                    return BLIND_SEARCH;
                case Algorithm::LocalSearch:
                    return LOCAL_SEARCH;
                default:
                    return "";
                    break;
            }
        }
    };
}

namespace mdata
{
    template<class T>
    class SearchAlgorithm
    {
    public:
        SearchAlgorithm() : timeDiff(0.0) {}
        virtual ~SearchAlgorithm() = 0;
        virtual TestResult<T> run(mfunc::mfuncPtr<T> funcPtr, const T fMin, const T fMax, Population<T>* const pop, const T alpha) = 0;
    protected:
        double timeDiff;
        high_resolution_clock::time_point timer;

        void startTimer()
        {
            timer = high_resolution_clock::now();
        }

        double stopTimer()
        {
            high_resolution_clock::time_point t_end = high_resolution_clock::now();
            return static_cast<double>(duration_cast<nanoseconds>(t_end - timer).count()) / 1000000.0;
        }
    };
}

// Trivial implementation of pure-virtual destructor
// as required by the C++ language
template<class T>
mdata::SearchAlgorithm<T>::~SearchAlgorithm() { }

#endif

// ========================
// End of searchalg.h
// ========================
```

7.21 include/stringutils.h File Reference

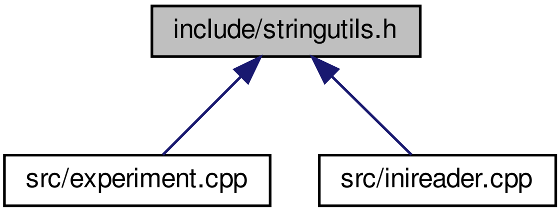
Contains various string manipulation helper functions.

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

Include dependency graph for stringutils.h:



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



Namespaces

- util

7.21.1 Detailed Description

Contains various string manipulation helper functions.

Author

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

Date

2019-04-01

Definition in file stringutils.h.

7.22 stringutils.h

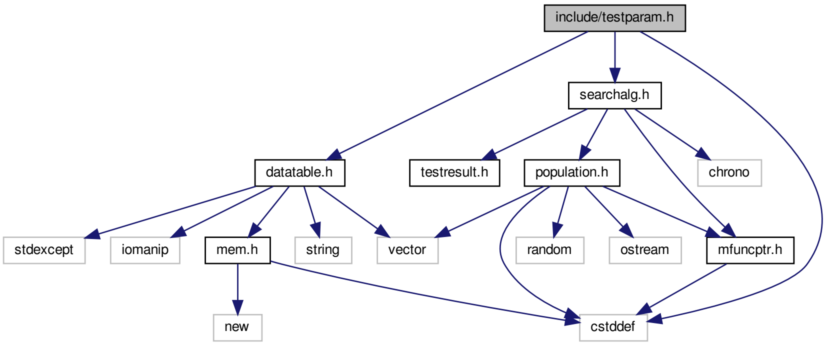
```

#ifndef __STRINGUTILS_H
#define __STRINGUTILS_H

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

namespace util
{
    // ======================================================
    // The string functions below were written by Evan Teran
    // from Stack Overflow:
    // https://stackoverflow.com/questions/216823/whats-the-best-way-to-trim-stdstring
    // ======================================================

    // trim from start (in place)
    static inline void s_ltrim(std::string &s) {
        s.erase(s.begin(), std::find_if(s.begin(), s.end(),
                std::not1(std::ptr_fun<int, int>(std::isspace))));
    }

    // trim from end (in place)
    static inline void s_rtrim(std::string &s) {
        s.erase(std::find_if(s.rbegin(), s.rend(),
                std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
    }

    // trim from both ends (in place)
    static inline void s_trim(std::string &s) {
        s_ltrim(s);
        s_rtrim(s);
    }

    // trim from start (copying)
    static inline std::string s_ltrim_copy(std::string s) {
        s_ltrim(s);
        return s;
    }

    // trim from end (copying)
    static inline std::string s_rtrim_copy(std::string s) {
        s_rtrim(s);
        return s;
    }

    // trim from both ends (copying)
    static inline std::string s_trim_copy(std::string s) {
        s_trim(s);
        return s;
    }
}
#endif

// ========================
// End of stringutils.h
// ========================
```

7.23 include/testparam.h File Reference

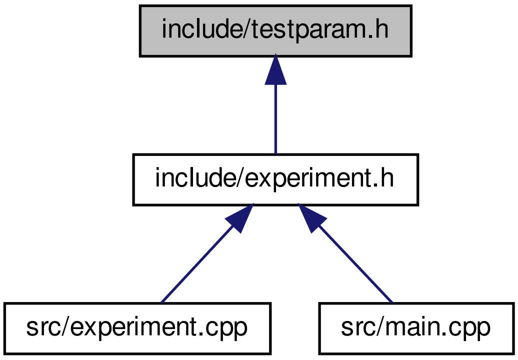
Contains the definition of the TestParameters struct, which is a data type used to transfer test parameters between functions.

```
#include <cstddef>
#include "datatable.h"
#include "searchalg.h"
```

Include dependency graph for testparam.h:



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



Classes

- struct mdata::TestParameters< T >

  Packs together various test experiment parameters.

Namespaces

- mdata

7.23.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-20

Copyright

Copyright (c) 2019

Definition in file testparam.h.

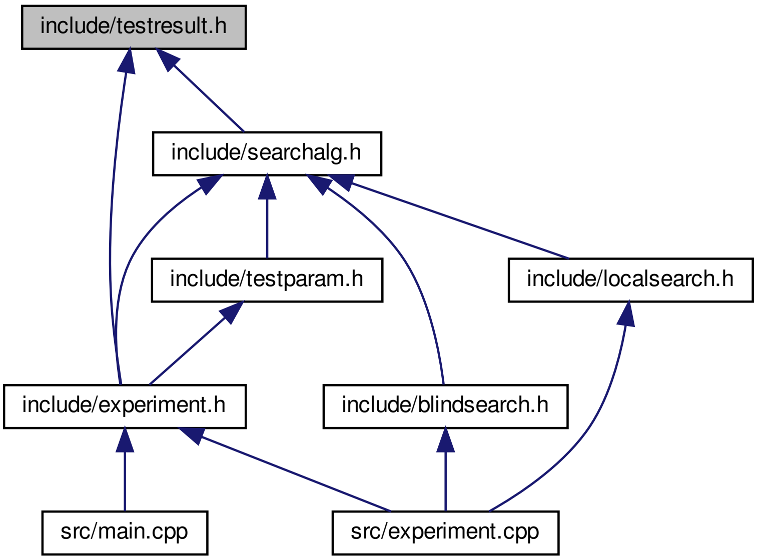
7.24 testparam.h

```
#ifndef __TESTPARAM_H
#define __TESTPARAM_H

#include <cstddef> // size_t definition
#include "datatable.h"
#include "searchalg.h"

namespace mdata
{
    template <class T>
    struct TestParameters
    {
        unsigned int funcId;
        T alpha;
        unsigned int resultsCol;
        unsigned int execTimesCol;
        size_t resultsRow;
        size_t execTimesRow;
        DataTable<T>* resultsTable;
        DataTable<T>* execTimesTable;
        enums::Algorithm alg;
        TestParameters()
        {
            funcId = 1;
            alpha = 0;
            alg = enums::Algorithm::BlindSearch;
            resultsTable = nullptr;
            execTimesTable = nullptr;
            resultsCol = 0;
            execTimesCol = 0;
            resultsRow = 0;
            execTimesRow = 0;
        }
    };
}

#endif

// ========================
// End of testparam.h
// ========================
```

7.25 include/testresult.h File Reference

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

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



Classes

- struct mdata::TestResult< T >

Namespaces

- mdata

7.25.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-19

Copyright

Copyright (c) 2019

Definition in file testresult.h.

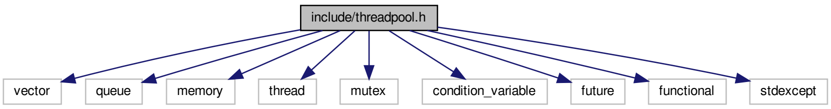
7.26 testresult.h

```
#ifndef __TESTRESULT_H
#define __TESTRESULT_H

namespace mdata
{
    template<class T>
    struct TestResult
    {
        const int err; // Error code. 0 = no error.
        const T fitness; // Fitness result
        const double execTime; // Algorithm execution time in miliseconds

        TestResult(int _err, T _fitness, double _execTime) : err(_err), fitness(_fitness), execTime(_execTime)
        {
        }
    };
} // mdata

#endif

// =========================
// End of testresult.h
// =========================
```

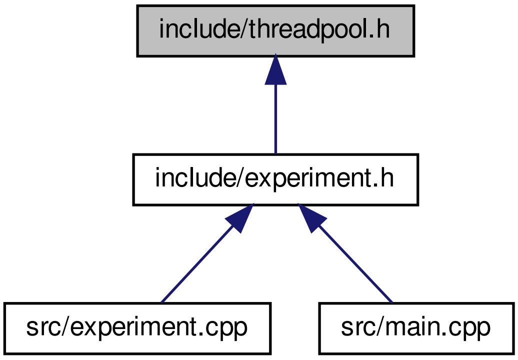
7.27 include/threadpool.h File Reference

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

Include dependency graph for threadpool.h:



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



Classes

- class ThreadPool

7.28 threadpool.h

```

#ifndef __THREADPOOL_H
#define __THREADPOOL_H

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

class ThreadPool {
public:
    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();
private:
    // need to keep track of threads so we can join them
    std::vector< std::thread > workers;
    // the task queue
    std::queue< std::function<void()> > tasks;

    // synchronization
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
};

// the constructor just launches some amount of workers
inline ThreadPool::ThreadPool(size_t threads)
    :   stop(false)
{
    for(size_t i = 0;i<threads;++i)
```

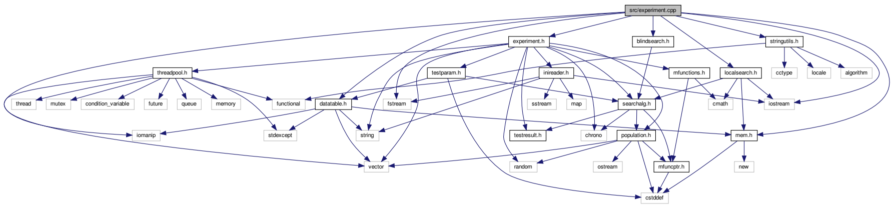
```
        workers.emplace_back(
            [this]
            {
                for(;;)
                {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        this->condition.wait(lock,
                            [this]{ return this->stop || !this->tasks.empty(); });
                        if(this->stop && this->tasks.empty())
                            return;
                        task = std::move(this->tasks.front());
                        this->tasks.pop();
                    }

                    task();
                }
            }
        );
}

// add new work item to the pool
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type>
{
    using return_type = typename std::result_of<F(Args...)>::type;

    auto task = std::make_shared< std::packaged_task<return_type()> >(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );

    std::future<return_type> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(queue_mutex);

        // don't allow enqueueing after stopping the pool
        if(stop)
            throw std::runtime_error("enqueue on stopped ThreadPool");

        tasks.emplace([task](){ (*task)(); });
    }
    condition.notify_one();
    return res;
}

// the destructor joins all threads
inline ThreadPool::~ThreadPool()
{
    stopAndJoinAll();
}

inline void ThreadPool::stopAndJoinAll()
{
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }

    condition.notify_all();
    for(std::thread &worker: workers)
        worker.join();
}

#endif

// =========================
// End of threadpool.h
// =========================
```

7.29 src/experiment.cpp File Reference

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

```
#include "blindsearch.h"
#include "localsearch.h"
#include "stringutils.h"
#include "mem.h"
```

Include dependency graph for experiment.cpp:



Macros

- #define INI_TEST_SECTION "test"
- #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_ALPHA "alpha"
- #define INI_TEST_ALGORITHM "algorithm"
- #define INI_TEST_RESULTSFILE "results_file"
- #define INI_TEST_EXECTIMESFILE "exec_times_file"

7.29.1 Macro Definition Documentation

7.29.1.1 INI_FUNC_RANGE_SECTION

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

Definition at line 25 of file experiment.cpp.

7.29.1.2 INI_TEST_ALGORITHM

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

Definition at line 31 of file experiment.cpp.

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

7.29.1.3 INI_TEST_ALPHA

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

Definition at line 30 of file experiment.cpp.

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

7.29.1.4 INI_TEST_DIMENSIONS

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

Definition at line 27 of file experiment.cpp.

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

7.29.1.5 INI_TEST_EXECTIMESFILE

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

Definition at line 33 of file experiment.cpp.

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

7.29.1.6 INI_TEST_ITERATIONS

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

Definition at line 28 of file experiment.cpp.

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

7.29.1.7 INI_TEST_NUMTHREADS

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

Definition at line 29 of file experiment.cpp.

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

7.29.1.8 INI_TEST_POPULATION

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

Definition at line 26 of file experiment.cpp.

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

7.29.1.9 INI_TEST_RESULTSFILE

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

Definition at line 32 of file experiment.cpp.

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

7.29.1.10 INI_TEST_SECTION

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

Definition at line 24 of file experiment.cpp.

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

7.30 experiment.cpp

```
00001
00013 #include <iostream>
00014 #include <fstream>
00015 #include <iomanip>
00016 #include "experiment.h"
00017 #include "datatable.h"
00018 #include "blindsearch.h"
00019 #include "localsearch.h"
00020 #include "stringutils.h"
00021 #include "mem.h"
00022
00023 // Ini file string sections and keys
00024 #define INI_TEST_SECTION "test"
00025 #define INI_FUNC_RANGE_SECTION "function_range"
00026 #define INI_TEST_POPULATION "population"
00027 #define INI_TEST_DIMENSIONS "dimensions"
00028 #define INI_TEST_ITERATIONS "iterations"
00029 #define INI_TEST_NUMTHREADS "num_threads"
00030 #define INI_TEST_ALPHA "alpha"
00031 #define INI_TEST_ALGORITHM "algorithm"
00032 #define INI_TEST_RESULTSFILE "results_file"
00033 #define INI_TEST_EXECTIMESFILE "exec_times_file"
00034
00035 using namespace std;
00036 using namespace std::chrono;
00037 using namespace mfunc;
00038
00042 template<class T>
00043 Experiment<T>::Experiment()
00044     : vBounds(nullptr), tPool(nullptr), resultsFile(""), execTimesFile(""), iterations(0)
00045 {
00046 }
00047
```

```
template<class T>
Experiment<T>::~Experiment()
{
    releaseThreadPool();
    releasePopulationPool();
    releaseVBounds();
}

template<class T>
bool Experiment<T>::init(const char* paramFile)
{
    try
    {
        // Open and parse parameters file
        if (!iniParams.openFile(paramFile))
        {
            cerr << "Experiment init failed: Unable to open param file: " << paramFile << endl;
            return false;
        }

        // Extract test parameters from ini file
        long numberSol = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_POPULATION);
        long numberDim = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_DIMENSIONS);
        long numberIter = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_ITERATIONS);
        long numberThreads = iniParams.getEntryAs<long>(INI_TEST_SECTION, INI_TEST_NUMTHREADS);
        alpha = iniParams.getEntryAs<T>(INI_TEST_SECTION, INI_TEST_ALPHA);
        unsigned int selectedAlg = iniParams.getEntryAs<unsigned int>(INI_TEST_SECTION, INI_TEST_ALGORITHM);
        resultsFile = iniParams.getEntry(INI_TEST_SECTION, INI_TEST_RESULTSFILE);
        execTimesFile = iniParams.getEntry(INI_TEST_SECTION, INI_TEST_EXECTIMESFILE);

        // Verify test parameters
        if (numberSol <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_POPULATION << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberDim <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_DIMENSIONS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberIter <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ITERATIONS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (numberThreads <= 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_NUMTHREADS << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }
        else if (alpha == 0)
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ALPHA << " is missing or is equal to zero: " << paramFile << endl;
            return false;
        }
        else if (selectedAlg >= static_cast<unsigned int>(enums::Algorithm::Count))
        {
            cerr << "Experiment init failed: Param file [test]->"
                << INI_TEST_ALGORITHM << " entry missing or out of bounds: " << paramFile << endl;
            return false;
        }

        // Cast iterations and test algorithm to correct types
        iterations = (size_t)numberIter;
        testAlg = static_cast<enums::Algorithm>(selectedAlg);

        // Print test parameters to console
        cout << "Population size: " << numberSol << endl;
        cout << "Dimensions: " << numberDim << endl;
        cout << "Iterations: " << iterations << endl;
        cout << "Alpha value: " << alpha << endl;
        cout << "Algorithm: " << enums::AlgorithmNames::get(testAlg) << endl;

        // Allocate memory for all population objects. We need one for each thread to prevent conflicts.
        if (!allocatePopulationPool((size_t)numberThreads, (size_t)numberSol, (size_t)numberDim))
        {
            cerr << "Experiment init failed: Unable to allocate populations." << endl;
            return false;
        }

        // Allocate memory for function vector bounds
        if (!allocateVBounds())
        {
            cerr << "Experiment init failed: Unable to allocate vector bounds array." << endl;
            return false;
        }

        // Fill function bounds array with data parsed from iniParams
        if (!parseFuncBounds())
        {
            cerr << "Experiment init failed: Unable to parse vector bounds array." << endl;
            return false;
        }

        // Allocate thread pool
        if (!allocateThreadPool((size_t)numberThreads))
        {
            cerr << "Experiment init failed: Unable to allocate thread pool." << endl;
            return false;
        }

        cout << "Started " << numberThreads << " worker threads ..." << endl;

        // Ready to run an experiment
        return true;
    }
    catch (const std::exception& ex)
    {
        cerr << "Exception occurred while initializing experiment: " << ex.what() << endl;
        return false;
    }
    catch (...)
    {
        cerr << "Unknown Exception occurred while initializing experiment." << endl;
        return false;
    }
}

template<class T>
int Experiment<T>::testAllFunc()
{
    if (populationsPool.size() == 0) return 1;

    // Construct results and execution times tables
    mdata::DataTable<T> resultsTable(iterations, (size_t)
NUM_FUNCTIONS);
    mdata::DataTable<T> execTimesTable(iterations, (size_t)NUM_FUNCTIONS);

    // Prepare thread futures vector, used to ensure all async tasks complete
    // succesfully.
    std::vector<std::future<int>> testFutures;

    // Start recording total execution time
    high_resolution_clock::time_point t_start = high_resolution_clock::now();

    // For each of the NUM_FUNCTIONS functions, prepare a TestParameters
    // struct and queue an asynchronous test that will be picked up and
    // executed by one of the threads in the thread pool.
    for (unsigned int i = 0; i < NUM_FUNCTIONS; i++)
    {
        // Update column labels for results and exec times tables
        resultsTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));
        execTimesTable.setColLabel((size_t)i, FunctionDesc::get(i + 1));

        // Queue up a new function test for each iteration
        for (size_t iter = 0; iter < iterations; iter++)
        {
            mdata::TestParameters<T> curParam;
            curParam.funcId = i + 1;
            curParam.alpha = alpha;
            curParam.alg = testAlg;
            curParam.resultsTable = &resultsTable;
            curParam.execTimesTable = &execTimesTable;
            curParam.resultsCol = i;
            curParam.execTimesCol = i;
            curParam.resultsRow = iter;
            curParam.execTimesRow = iter;

            // Add function test to async queue
            testFutures.emplace_back(
                tPool->enqueue(&Experiment<T>::testFuncThreaded, this, curParam)
            );
        }
    }

    // Get the total number of async tasks queued
    const double totalFutures = static_cast<double>(testFutures.size());
    int tensPercentile = -1;
    std::chrono::microseconds waitTime(100);

    // Loop until all async tasks are completed and the thread futures
    // array is empty
    while (testFutures.size() > 0)
    {
        // Sleep a little bit since the async thread tasks are higher priority
        std::this_thread::sleep_for(waitTime);

        // Get iterator to first thread future
        auto it = testFutures.begin();

        // Loop through all thread futures
        while (it != testFutures.end())
        {
            if (!it->valid())
            {
                // An error occured with one of the threads
                cerr << "Error: Thread future invalid.";
                tPool->stopAndJoinAll();
                return 1;
            }

            // Get the status of the current thread future (async task)
            std::future_status status = it->wait_for(waitTime);
            if (status == std::future_status::ready)
            {
                // Task has completed, get return value
                int errCode = it->get();
                if (errCode)
                {
                    // An error occurred while running the task.
                    // Bail out of function
                    tPool->stopAndJoinAll();
                    return errCode;
                }

                // Remove processed task future from vector
                it = testFutures.erase(it);

                // Calculate the percent completed of all tasks, rounded to the nearest 10%
                int curPercentile = static_cast<int>(((totalFutures - testFutures.size()) / totalFutures) * 10);
                if (curPercentile > tensPercentile)
                {
                    // Print latest percent value to the console
                    tensPercentile = curPercentile;
                    cout << "~" << (tensPercentile * 10) << "% " << flush;
                }
            }
            else
            {
                // Async task has not yet completed, advance to the next one
                it++;
            }
        }
    }

    // Record total execution time and print it to the console
    high_resolution_clock::time_point t_end = high_resolution_clock::now();
    long double totalExecTime = static_cast<long double>(duration_cast<nanoseconds>(t_end - t_start).count()) / 1000000000.0L;

    cout << endl << "Test finished. Total time: " << std::setprecision(7) << totalExecTime << " seconds." << endl;

    if (!resultsFile.empty())
    {
        // Export results table to a *.csv file
        cout << "Exporting results to: " << resultsFile << endl;
        resultsTable.exportCSV(resultsFile.c_str());
    }

    if (!execTimesFile.empty())
    {
        // Export exec times table to a *.csv file
        cout << "Exporting execution times to: " << execTimesFile << endl;
        execTimesTable.exportCSV(execTimesFile.c_str());
    }

    cout << flush;

    return 0;
}

template<class T>
int Experiment<T>::testFuncThreaded(mdata::TestParameters<T> tParams)
{
    mdata::SearchAlgorithm<T>* alg;

    // Construct a search algorithm object for the selected alg
    switch (tParams.alg)
    {
        case enums::Algorithm::BlindSearch:
            alg = new mdata::BlindSearch<T>();
            break;
        case enums::Algorithm::LocalSearch:
            alg = new mdata::LocalSearch<T>();
            break;
        default:
            cerr << "Invalid algorithm selected." << endl;
            return 1;
    }

    // Retrieve the function bounds
    const RandomBounds<T>& funcBounds = vBounds[tParams.funcId - 1];

    // Retrieve the next available population object from the population pool
    mdata::Population<T>* pop = popPoolRemove();

    // Run the search algorithm one and record the results
    auto tResult = alg->run(Functions<T>::get(tParams.funcId), funcBounds.min, funcBounds.max, pop, tParams.alpha);

    // Place the population object back into the pool to be reused by anther thread
    popPoolAdd(pop);

    if (tResult.err)
    {
        cerr << "Error while testing function " << tParams.funcId << endl;
        return tResult.err;
    }

    // Update results table and execution times table with algorithm results
    tParams.resultsTable->setEntry(tParams.resultsRow, tParams.resultsCol, tResult.fitness);
    tParams.execTimesTable->setEntry(tParams.execTimesRow, tParams.execTimesCol, tResult.execTime);

    delete alg;
    return 0;
}

template<class T>
mdata::Population<T>* Experiment<T>::popPoolRemove()
{
    mdata::Population<T>* retPop = nullptr;
    std::chrono::microseconds waitTime(10);

    while (true)
    {
        {
            std::lock_guard<std::mutex> lk(popPoolMutex);
            if (populationsPool.size() > 0)
            {
                retPop = populationsPool.back();
                populationsPool.pop_back();
            }
        }

        if (retPop != nullptr)
            return retPop;
        else
            std::this_thread::sleep_for(waitTime);
    }
}

```

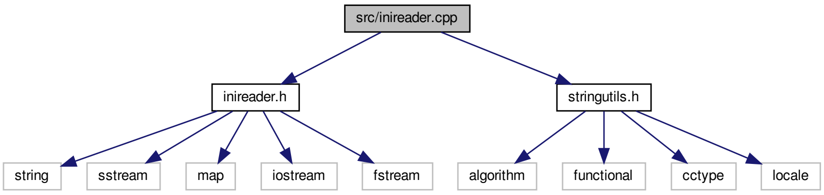
```
template<class T>
void Experiment<T>::popPoolAdd(mdata::Population<T>* popPtr)
{
    if (popPtr == nullptr) return;

    std::lock_guard<std::mutex> lk(popPoolMutex);

    populationsPool.push_back(popPtr);
}

template<class T>
bool Experiment<T>::parseFuncBounds()
{
    if (vBounds == nullptr) return false;

    const string delim = ",";
    const string section = "function_range";
    string s_min;
    string s_max;

    // Extract the bounds for each function
    for (unsigned int i = 1; i <= NUM_FUNCTIONS; i++)
    {
        // Get bounds entry from ini file for current function
        string entry = iniParams.getEntry(section, to_string(i));
        if (entry.empty())
        {
            cerr << "Error parsing bounds for function: " << i << endl;
            return false;
        }

        // Find index of ’,’ delimeter in entry string
        auto delimPos = entry.find(delim);
        if (delimPos == string::npos || delimPos >= entry.length() - 1)
        {
            cerr << "Error parsing bounds for function: " << i << endl;
            return false;
        }

        // Split string and extract min/max strings
        s_min = entry.substr((size_t)0, delimPos);
        s_max = entry.substr(delimPos + 1, entry.length());
        util::s_trim(s_min);
        util::s_trim(s_max);

        // Attempt to parse min and max strings into double values
        try
        {
            RandomBounds<T>& b = vBounds[i - 1];
            b.min = atof(s_min.c_str());
            b.max = atof(s_max.c_str());
        }
        catch(const std::exception& e)
        {
            cerr << "Error parsing bounds for function: " << i << endl;
            std::cerr << e.what() << ’\n’;
            return false;
        }
    }

    return true;
}

template<class T>
bool Experiment<T>::allocatePopulationPool(size_t count, size_t popSize, size_t dimensions)
{
    releasePopulationPool();

    std::lock_guard<std::mutex> lk(popPoolMutex);

    try
    {
        for (int i = 0; i < count; i++)
        {
            auto newPop = new(std::nothrow) mdata::Population<T>(popSize, dimensions);
            if (newPop == nullptr)
            {
                std::cerr << "Error allocating populations." << ’\n’;
                return false;
            }

            populationsPool.push_back(newPop);
        }

        return true;
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << '\n';
        return false;
    }
}

template<class T>
void Experiment<T>::releasePopulationPool()
{
    std::lock_guard<std::mutex> lk(popPoolMutex);

    if (populationsPool.size() == 0) return;

    for (int i = 0; i < populationsPool.size(); i++)
    {
        if (populationsPool[i] != nullptr)
        {
            delete populationsPool[i];
            populationsPool[i] = nullptr;
        }
    }

    populationsPool.clear();
}

template<class T>
bool Experiment<T>::allocateVBounds()
{
    vBounds = util::allocArray<RandomBounds<T>>(NUM_FUNCTIONS);
    return vBounds != nullptr;
}

template<class T>
void Experiment<T>::releaseVBounds()
{
    if (vBounds == nullptr) return;

    util::releaseArray<RandomBounds<T>>(vBounds);
}

template<class T>
bool Experiment<T>::allocateThreadPool(size_t numThreads)
{
    releaseThreadPool();

    tPool = new(std::nothrow) ThreadPool(numThreads);
    return tPool != nullptr;
}

template<class T>
void Experiment<T>::releaseThreadPool()
{
    if (tPool == nullptr) return;

    delete tPool;
    tPool = nullptr;
}

// Explicit template specializations due to separate implementations in this CPP file
template class mfunc::Experiment<float>;
template class mfunc::Experiment<double>;
template class mfunc::Experiment<long double>;

// =========================
// End of experiment.cpp
// =========================
```

7.31 src/inireader.cpp File Reference

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

```
#include "inireader.h"
#include "stringutils.h"
```

Include dependency graph for inireader.cpp:



7.31.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.1

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file inireader.cpp.

7.32 inireader.cpp

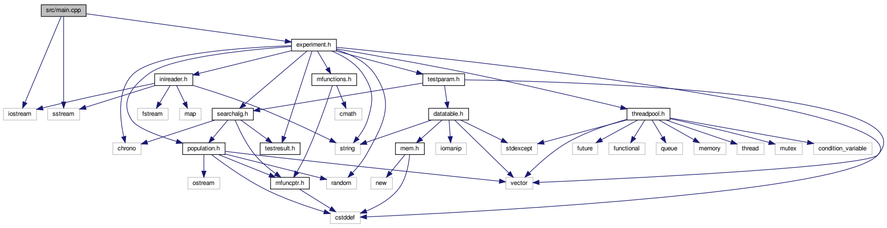
```

#include "inireader.h"
#include "stringutils.h"

using namespace util;

IniReader::IniReader() : file(""), iniMap()
{
}

IniReader::~IniReader()
{
    iniMap.clear();
}

bool IniReader::openFile(std::string filePath)
{
    file = filePath;
    if (!parseFile())
        return false;
    return true;
}

bool IniReader::sectionExists(std::string section)
{
    return iniMap.find(section) != iniMap.end();
}

bool IniReader::entryExists(std::string section, std::string entry)
{
    auto it = iniMap.find(section);
    if (it == iniMap.end()) return false;

    return it->second.find(entry) != it->second.end();
}

std::string IniReader::getEntry(std::string section, std::string entry)
{
    if (!entryExists(section, entry)) return std::string();

    return iniMap[section][entry];
}

bool IniReader::parseFile()
{
    iniMap.clear();

    using namespace std;

    ifstream inputF(file, ifstream::in);
    if (!inputF.good()) return false;

    string curSection;
    string line;

    while (getline(inputF, line))
    {
        // Trim whitespace on both ends of the line
        s_trim(line);

        // Ignore empty lines and comments
        if (line.empty() || line.front() == '#')
        {
            continue;
        }
        else if (line.front() == '[' && line.back() == ']')
        {
            // Line is a section definition
            // Erase brackets and trim to get section name
            line.erase(0, 1);
            line.erase(line.length() - 1, 1);
            s_trim(line);
            curSection = line;
        }
        else if (!curSection.empty())
        {
            // Line is an entry, parse the key and value
            parseEntry(curSection, line);
        }
    }

    // Close input file
    inputF.close();
    return true;
}

void IniReader::parseEntry(const std::string& sectionName, const std::string& entry)
{
    using namespace std;

    // Split string around equals sign character
    const string delim = "=";
    string entryName;
    string entryValue;

    // Find index of '='
    auto delimPos = entry.find(delim);

    if (delimPos == string::npos || delimPos >= entry.length() - 1)
        return; // '=' is missing, or is last char in string

    // Extract entry name/key and value
    entryName = entry.substr((size_t)0, delimPos);
    entryValue = entry.substr(delimPos + 1, entry.length());

    // Remove leading and trailing whitespace
    s_trim(entryName);
    s_trim(entryValue);

    // We cannot have entries with empty keys
    if (entryName.empty()) return;

    // Add entry to cache
    iniMap[sectionName][entryName] = entryValue;
}

// =========================
// End of inireader.cpp
// =========================
```

7.33 src/main.cpp File Reference

Program entry point. Creates and runs CS471 project 2 experiment.

```
#include <iostream>
#include <sstream>
#include "experiment.h"
```

Include dependency graph for main.cpp:



Functions

- template<class T >
  int runExp (const char ∗paramFile)

  Runs the experiment using the given data type and parameter file. Currently supports three different data types: float, double, and long double.
- int main (int argc, char ∗∗argv)

7.33.1 Detailed Description

Program entry point. Creates and runs CS471 project 2 experiment.

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-01

Copyright

Copyright (c) 2019

Definition in file main.cpp.

7.33.2 Function Documentation

7.33.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

Definition at line 46 of file main.cpp.

```
{
   // Make sure we have enough command line args
   if (argc <= 1)
   {
      cout << "Error: Missing command line parameter." << endl;
      cout << "Proper usage: " << argv[0] << " [param file]" << endl;
      return EXIT_FAILURE;
   }

   // Default data type is double
   int dataType = 1;

   // User specified a data type, retrieve the value
   if (argc > 2)
   {
      std::stringstream ss(argv[2]);
      ss >> dataType;
      if (!ss) dataType = 1;
   }

   // Verify specified data type switch
   if (dataType < 0 || dataType > 2)
   {
      cout << dataType << " is not a valid data type index. Value must be between 0 and 2." << endl;
      dataType = 1;
   }

   // Run experiment with correct data type and return success code
   switch (dataType)
   {
      case 0:
         return runExp<float>(argv[1]);
      case 1:
         return runExp<double>(argv[1]);
      case 2:
         return runExp<long double>(argv[1]);
      default:
         return EXIT_FAILURE;
   }
}
```

7.33.2.2 runExp()

```
template<class T >
int runExp (
              const char * paramFile )
```

Runs the experiment using the given data type and parameter file. Currently supports three different data types: float, double, and long double.

Template Parameters

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 project 1 experiment class
00032    mfunc::Experiment<T> ex;
00033
00034    // Print size of selected data type in bits
00035    cout << "Float size: " << (sizeof(T) * 8) << "-bits" << endl;
00036    cout << "Input parameters file: " << paramFile << endl;
00037    cout << "Initializing experiment ..." << endl;
00038
00039    // If experiment initialization fails, return failure
00040    if (!ex.init(paramFile))
00041       return EXIT_FAILURE;
00042    else
00043       return ex.testAllFunc();
00044 }
```

7.34 main.cpp

```
00001
00013 #include <iostream>
00014 #include <sstream>
00015 #include "experiment.h"
00016
00017 using namespace std;
00018
00028 template<class T>
00029 int runExp(const char* paramFile)
00030 {
00031    // Create an instance of the project 1 experiment class
00032    mfunc::Experiment<T> ex;
00033
00034    // Print size of selected data type in bits
00035    cout << "Float size: " << (sizeof(T) * 8) << "-bits" << endl;
00036    cout << "Input parameters file: " << paramFile << endl;
00037    cout << "Initializing experiment ..." << endl;
00038
00039    // If experiment initialization fails, return failure
00040    if (!ex.init(paramFile))
00041       return EXIT_FAILURE;
00042    else
00043       return ex.testAllFunc();
00044 }
00045
00046 int main(int argc, char** argv)
00047 {
00048    // Make sure we have enough command line args
00049    if (argc <= 1)
00050    {
00051       cout << "Error: Missing command line parameter." << endl;
00052       cout << "Proper usage: " << argv[0] << " [param file]" << endl;
00053       return EXIT_FAILURE;
00054    }
00055
```

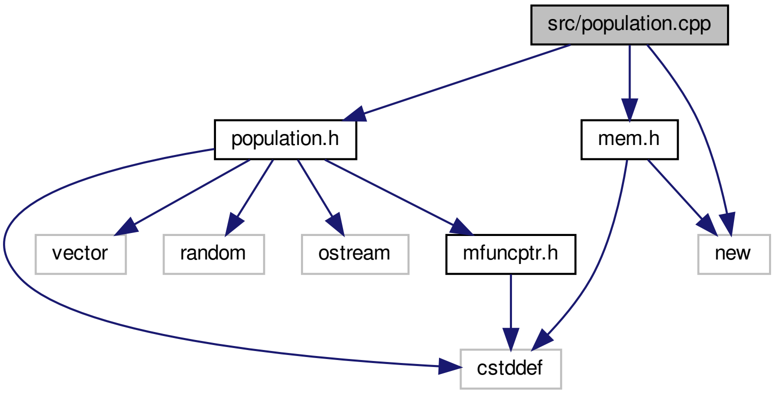
```
   // Default data type is double
   int dataType = 1;

   // User specified a data type, retrieve the value
   if (argc > 2)
   {
      std::stringstream ss(argv[2]);
      ss >> dataType;
      if (!ss) dataType = 1;
   }

   // Verify specified data type switch
   if (dataType < 0 || dataType > 2)
   {
      cout << dataType << " is not a valid data type index. Value must be between 0 and 2." << endl;
      dataType = 1;
   }

   // Run experiment with correct data type and return success code
   switch (dataType)
   {
      case 0:
         return runExp<float>(argv[1]);
      case 1:
         return runExp<double>(argv[1]);
      case 2:
         return runExp<long double>(argv[1]);
      default:
         return EXIT_FAILURE;
   }
}

// ========================
// End of main.cpp
// ========================
```

7.35 src/population.cpp File Reference

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

```
#include "population.h"
#include "mem.h"
#include <new>
```

Include dependency graph for population.cpp:



7.35.1 Detailed Description

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

Author

Andrew Dunn (Andrew.Dunn@cwu.edu)

Version

0.2

Date

2019-04-04

Copyright

Copyright (c) 2019

Definition in file population.cpp.

7.36 population.cpp

```

#include "population.h"
#include "mem.h"
#include <new>

using namespace mdata;
using namespace util;

template <class T>
Population<T>::Population(size_t pSize, size_t dimensions) : popMatrix(nullptr), popSize(pSize), popDim(dimensions)
{
    if (!allocPopMatrix() || !allocPopFitness())
        throw std::bad_alloc();
}

template <class T>
Population<T>::~Population()
{
    releasePopMatrix();
    releasePopFitness();
}

template <class T>
bool Population<T>::isReady()
{
    return popMatrix != nullptr && popFitness != nullptr;
}

template <class T>
size_t Population<T>::getPopulationSize()
{
    return popSize;
}

template <class T>
size_t Population<T>::getDimensionsSize()
{
    return popDim;
}

template <class T>
T* Population<T>::getPopulationPtr(size_t popIndex)
{
    if (popFitness == nullptr || popIndex >= popSize) return nullptr;

    return popMatrix[popIndex];
}

template <class T>
```

```
bool Population<T>::generate(T minBound, T maxBound)
{
    if (popMatrix == nullptr) return false;

    // Generate a new seed for the mersenne twister engine
    rgen = std::mt19937(rdev());

    // Set up a normal (bell-shaped) distribution for the random number generator with the correct function bounds
    std::uniform_real_distribution<double> dist((double)minBound, (double)maxBound);

    // Generate values for all vectors in popMatrix
    for (size_t s = 0; s < popSize; s++)
    {
        for (size_t d = 0; d < popDim; d++)
        {
            T rand = (T)dist(rgen);
            popMatrix[s][d] = rand;
        }
    }

    // Reset popFitness values to 0
    initArray<T>(popFitness, popSize, (T)0.0);

    return true;
}

template<class T>
bool Population<T>::setFitness(size_t popIndex, T value)
{
    if (popFitness == nullptr || popIndex >= popSize) return false;

    popFitness[popIndex] = value;

    return true;
}

template<class T>
bool Population<T>::calcFitness(size_t popIndex, mfunc::mfuncPtr<T> funcPtr)
{
    if (popFitness == nullptr || popIndex >= popSize) return false;

    popFitness[popIndex] = funcPtr(popMatrix[popIndex], popDim);

    return true;
}

template<class T>
T Population<T>::getFitness(size_t popIndex)
{
    if (popFitness == nullptr || popIndex >= popSize) return 0;

    return popFitness[popIndex];
}

template<class T>
T* Population<T>::getFitnessPtr(size_t popIndex)
{
    if (popFitness == nullptr || popIndex >= popSize) return 0;

    return &popFitness[popIndex];
}

template<class T>
std::vector<T> Population<T>::getAllFitness()
{
    return std::vector<T>(popFitness[0], popFitness[popSize]);
}

template<class T>
T* Population<T>::getBestFitnessPtr()
{
    return &popFitness[getBestFitnessIndex()];
}

template<class T>
size_t Population<T>::getBestFitnessIndex()
{
    size_t bestIndex = 0;

    for (size_t i = 1; i < popSize; i++)
    {
        if (popFitness[i] < popFitness[bestIndex])
            bestIndex = i;
    }

    return bestIndex;
}

template<class T>
void Population<T>::outputPopulation(std::ostream& outStream, const char* delim, const char* lineBreak)
{
    if (popMatrix == nullptr) return;

    for (size_t j = 0; j < popSize; j++)
    {
        for (size_t k = 0; k < popDim; k++)
        {
            outStream << popMatrix[j][k];
            if (k < popDim - 1)
                outStream << delim;
        }

        outStream << lineBreak;
    }
}

template<class T>
void Population<T>::outputFitness(std::ostream& outStream, const char* delim, const char* lineBreak)
{
    if (popFitness == nullptr) return;

    for (size_t j = 0; j < popSize; j++)
    {
        outStream << popFitness[j];
            if (j < popSize - 1)
                outStream << delim;
    }

    if (lineBreak != nullptr)
        outStream << lineBreak;
}

template <class T>
bool Population<T>::allocPopMatrix()
{
    if (popSize == 0 || popDim == 0) return false;

    popMatrix = allocMatrix<T>(popSize, popDim);
    initMatrix<T>(popMatrix, popSize, popDim, 0);

    return popMatrix != nullptr;
}

template <class T>
void Population<T>::releasePopMatrix()
{
    releaseMatrix<T>(popMatrix, popSize);
}

template <class T>
bool Population<T>::allocPopFitness()
{
    if (popSize == 0 || popDim == 0) return false;

    popFitness = allocArray<T>(popSize);
    initArray<T>(popFitness, popSize, 0);

    return popFitness != nullptr;
}

template <class T>
void Population<T>::releasePopFitness()
{
    releaseArray<T>(popFitness);
}

// Explicit template specializations due to separate implementations in this CPP file
template class mdata::Population<float>;
template class mdata::Population<double>;
template class mdata::Population<long double>;

// =========================
// End of population.cpp
// =========================
```

Index