Root-Finding Algorithms Solver
1.0

Generated by Doxygen 1.12.0

1 Root-Finding Algorithms Solver	1
1.0.1 Key Features	1
1.0.2 How to Use	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation	7
4.1 RootInfo Struct Reference	7
4.1.1 Member Data Documentation	7
4.1.1.1 decimal_places	7
4.1.1.2 iterations	7
4.1.1.3 root	7
5 File Documentation	9
5.1 functions.cpp File Reference	9
5.1.1 Function Documentation	9
5.1.1.1 f()	9
5.1.1.2 f_prime()	10
5.2 functions.h File Reference	10
5.2.1 Function Documentation	10
5.2.1.1 f()	10
5.2.1.2 f_prime()	10
5.3 functions.h	11
5.4 main.cpp File Reference	11
5.4.1 Detailed Description	12
5.4.2 Function Documentation	12
5.4.2.1 main()	12
5.5 methods.cpp File Reference	13
5.5.1 Function Documentation	14
5.5.1.1 bisection()	14
5.5.1.2 brent_method()	15
5.5.1.3 hybrid_method()	16
5.5.1.4 newton_raphson()	17
5.5.1.5 ridder_method()	17
5.6 methods.h File Reference	18
5.6.1 Function Documentation	19
5.6.1.1 bisection()	19
5.6.1.2 brent_method()	20
5.6.1.3 hybrid_method()	21
5.6.1.4 newton_raphson()	22

Index

5.6.1.5 ridder_method()	22
5.7 methods.h	23
5.8 plotting.cpp File Reference	24
5.8.1 Function Documentation	24
5.8.1.1 plot_function()	24
5.9 plotting.h File Reference	26
5.9.1 Function Documentation	26
5.9.1.1 plot_function()	26
5.10 plotting.h	27
5.11 utils.cpp File Reference	28
5.11.1 Function Documentation	29
5.11.1.1 calculate_decimal_places()	29
5.11.1.2 compare_all_methods()	29
5.11.1.3 get_user_input()	30
5.11.1.4 run_method()	31
5.11.1.5 run_method_user_selection()	31
5.11.1.6 run_problem_steps()	32
5.11.2 Variable Documentation	33
5.11.2.1 summary	33
5.12 utils.h File Reference	33
5.12.1 Function Documentation	34
5.12.1.1 calculate_decimal_places()	34
5.12.1.2 compare_all_methods()	35
5.12.1.3 get_user_input()	36
5.12.1.4 run_method()	37
5.12.1.5 run_method_user_selection()	37
5.12.1.6 run_problem_steps()	37
5.12.2 Variable Documentation	39
5.12.2.1 summary	39
5.13 utils.h	39

41

Chapter 1

Root-Finding Algorithms Solver

Using various numerical methods

The program allows users to choose from the following root-finding algorithms:

- · Bisection Method
- · Hybrid Method
- · Brent Method
- · Ridder Method
- · Newton-Raphson Method

Users can either use default parameters or customize them according to their needs. The program outputs include the root of the function, the number of iterations, and detailed steps.

Additionally, the program offers options to compare all the methods and display their performance side by side.

1.0.1 Key Features

- · Implements five distinct root-finding algorithms
- · Interactive user interface for method selection
- · Customizable parameters such as tolerance and initial guesses
- · Displays detailed performance metrics and results
- · Provides comparative analysis across the algorithms

1.0.2 How to Use

- 1. Run the program.
- 2. Select a root-finding algorithm or choose to compare all methods.
- 3. Enter custom parameters or use the default values.
- 4. View the results and performance metrics.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:	
RootInfo	7

4 Class Index

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

functions.cpp)																																							9
functions.h																																								10
main.cpp																																								
The	e n	nai	n (er	ıtr	У	00	in	t f	or	th	ie	R	00	ot-	Fi	ind	diı	ng	Α	ιlg	or	ith	nn	าร	S	ol	ve	r	pr	oje	ect								- 11
methods.cpp)																																							13
methods.h .																																								18
plotting.cpp																																								24
plotting.h .																																								26
utils.cpp																																								28
utils.h																																								33

6 File Index

Chapter 4

Class Documentation

4.1 RootInfo Struct Reference

#include <methods.h>

Public Attributes

- · long double root
- · int iterations
- int decimal_places

4.1.1 Member Data Documentation

4.1.1.1 decimal_places

int RootInfo::decimal_places

4.1.1.2 iterations

int RootInfo::iterations

4.1.1.3 root

long double RootInfo::root

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

• methods.h

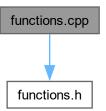
8 Class Documentation

Chapter 5

File Documentation

5.1 functions.cpp File Reference

```
#include "functions.h"
Include dependency graph for functions.cpp:
```



Functions

- long double f (long double x)
- long double f_prime (long double x)

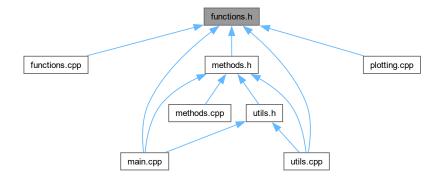
5.1.1 Function Documentation

5.1.1.1 f()

5.1.1.2 f_prime()

5.2 functions.h File Reference

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



Functions

- long double f (long double x)
- long double f_prime (long double x)

5.2.1 Function Documentation

5.2.1.1 f()

5.2.1.2 f_prime()

5.3 functions.h

5.3 functions.h

Go to the documentation of this file.

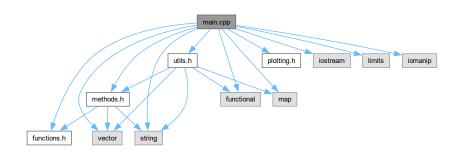
```
00001 /*
00002 @Author: Gilbert Young
00003 @Time: 2024/09/19 01:47
00004 @File_name: functions.h
00005 @IDE: VSCode
00006 @Formatter: Clang-Format
00007 @Description: Declaration of the function f(x) and its derivative f'(x).
00008 */
00009
0010 #ifndef FUNCTIONS_H
00011 #define FUNCTIONS_H
00012
00013 long double f(long double x);
00014 long double f_prime(long double x);
00015
00016 #endif // FUNCTIONS_H
```

5.4 main.cpp File Reference

The main entry point for the Root-Finding Algorithms Solver project.

```
#include "functions.h"
#include "methods.h"
#include "plotting.h"
#include "utils.h"
#include <iostream>
#include <limits>
#include <map>
#include <vector>
#include <functional>
#include <string>
#include <iomanip>
```

Include dependency graph for main.cpp:



Functions

• int main ()

5.4.1 Detailed Description

The main entry point for the Root-Finding Algorithms Solver project.

Author

Gilbert Young

Date

2024/09/19

5.4.2 Function Documentation

5.4.2.1 main()

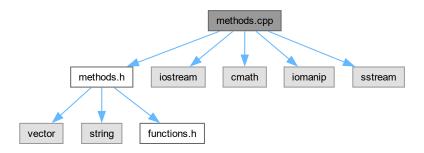
```
int main ()
00055 {
00056
          // Plot the function once at the beginning with range [-3, 3]
         plot_function(-3.0L, 3.0L, -10.0L, 10.0L);
00057
00058
00059
         char choice;
00060
00061
00062
             long double a = 0.0L, b = 0.0L, x0 = 0.0L, tol = 1e-14L;
00063
             std::string method_name;
00064
00065
             // Get user input for method selection
00066
             get_user_input(a, b, x0, method_name, tol);
00067
00068
             int max_iter = 1000;
00069
             //\ \mbox{Map} of methods excluding Newton-Raphson and Compare All Methods
00070
             std::map<std::string, std::function<long double(long double, long double, long double, int,
00071
     std::vector<std::string> &, int)» methods = {
00072
                 {"Bisection Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00073
00074
                      return bisection(a, b, tol, max_iter, iterations, decimal_places);
00075
                  }},
                 {"Hybrid Method", [](long double a, long double b, long double tol, int max_iter,
00076
     std::vector<std::string> &iterations, int decimal_places) -> long double
00077
00078
                      return hybrid_method(a, b, tol, max_iter, iterations, decimal_places);
00079
                  }}.
                 {"Brent Method", [](long double a, long double b, long double tol, int max_iter,
08000
     std::vector<std::string> &iterations, int decimal_places) -> long double
00081
00082
                      return brent_method(a, b, tol, max_iter, iterations, decimal_places);
00083
00084
                 {"Ridder Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00085
00086
                      return ridder_method(a, b, tol, max_iter, iterations, decimal_places);
00087
00088
00089
             if (method_name == "Newton-Raphson Method")
00090
00091
                 std::vector<std::string> iterations;
                 long double root = newton_raphson(x0, tol, max_iter, iterations,
00092
     calculate_decimal_places(tol));
00093
                 RootInfo info{root, static_cast<int>(iterations.size()), calculate_decimal_places(tol));
00094
                 summary[method_name].emplace_back(info);
00095
                 // Display results
00096
                 std::cout « "\nMethod: " « method_name « "\n";
00097
                 std::cout « "Initial guess: x0 = " « std::fixed « std::setprecision(info.decimal_places) «
     x0 « "\n";
00099
                 std::cout « "Iterations:\n";
00100
00101
                 for (const auto &iter : iterations)
00102
00103
                     std::cout « iter « "\n";
```

```
00104
00105
                   std::cout « "Iterations Count: " « iterations.size() « "\n";
00106
00107
               else if (method_name == "Problem Steps Mode")
00108
00109
                    // Run the problem steps
00110
                   run_problem_steps();
00111
00112
               else if (method_name == "Compare All Methods")
00113
               {
                   \ensuremath{//} Run the comparison
00114
                   compare_all_methods();
00115
00116
00117
00118
00119
                   \ensuremath{//} Get the method function
00120
                   auto it = methods.find(method name);
                   if (it != methods.end() && it->second != nullptr)
00121
00122
00123
                        run_method_user_selection(method_name, it->second, a, b, tol, max_iter);
00124
00125
                   else
00126
                   {
00127
                        std::cerr « "Method not found or not implemented.\n";
00128
                   }
00129
               }
00130
               // Output summary of all results
if (method_name != "Problem Steps Mode" && method_name != "Compare All Methods")
00131
00132
00133
00134
                   std::cout « "\n--- Summary of All Results ---\n";
00135
                   for (const auto &method : summary)
00136
00137
                        std::cout « "\nMethod: " « method.first « "\n";
00138
                        int idx = 1;
                        for (const auto &info : method.second)
00139
00140
                            std::cout « " Root " « idx++ « ": " « std::fixed «
00141
      std::setprecision(info.decimal_places) « info.root
00142
                                      «" | Iterations: " « info.iterations « "\n";
00143
00144
                   // Clear summary for next run
00145
00146
                   summary.clear();
00147
00148
               // Ask user if they want to run again std::cout \alpha "\nDo you want to run the program again? (y/n): ";
00149
00150
               std::cin » choice;
00151
00152
00153
          } while (choice == 'y' || choice == 'Y');
00154
00155
          // Pause and wait for user input before exiting
          std::cout « "\nPress Enter to exit...";
00156
          \verb|std::cin.ignore(std::numeric_limits < std::streamsize > :: max(), \ ' \ ');|
00157
00158
          std::cin.get();
00160
           return 0;
00161 }
```

5.5 methods.cpp File Reference

```
#include "methods.h"
#include <iostream>
#include <cmath>
#include <iomanip>
#include <sstream>
```

Include dependency graph for methods.cpp:



Functions

- long double bisection (long double a, long double b, long double tol, int max_iter, std::vector< std::string >
 &iterations, int decimal_places)
- long double newton_raphson (long double x0, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double hybrid_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double brent_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double ridder_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)

5.5.1 Function Documentation

5.5.1.1 bisection()

```
long double bisection (
               long double a,
               long double b,
              long double tol,
               int max_iter,
               std::vector< std::string > & iterations,
               int decimal_places)
00018 {
00019
          long double fa = f(a), fb = f(b);
00020
          if (fa * fb >= 0)
00021
              std::cerr \mbox{``Bisection method fails. f(a)} and f(b) should have opposite signs.\n";
00022
00023
              return NAN;
00024
00025
00026
          long double c = a;
          for (int i = 0; i < max_iter; ++i)</pre>
00027
00028
              c = (a + b) / 2.0L;
00029
              long double fc = f(c);
00030
00031
              std::ostringstream oss;
00032
              oss « "Step " « i + 1 « ": [" « std::fixed « std::setprecision(decimal_places) « a « ", " « b
      « "]";
00033
              iterations.push_back(oss.str());
00034
              if ((b - a) / 2.0L < tol)
00035
                  break;
              if (fa * fc < 0)</pre>
00036
```

```
00037
              {
                  b = c;
00038
00039
                  fb = fc;
00040
00041
              else
00042
              {
                  a = c;
00043
00044
                  fa = fc;
00045
00046
00047
          return c;
00048 }
```

5.5.1.2 brent_method()

```
long double brent_method (
               long double a,
               long double b,
               long double tol,
               int max_iter,
               std::vector< std::string > & iterations,
               int decimal_places)
00140 {
00141
          long double fa = f(a), fb = f(b);
00142
          if (fa * fb >= 0)
00143
              std::cerr « "Brent's method fails. f(a) and f(b) should have opposite signs.\n";
00144
00145
              return NAN;
00146
          }
00147
00148
          if (fabs(fa) < fabs(fb))</pre>
00149
00150
              std::swap(a, b);
00151
              std::swap(fa, fb);
00152
          }
00153
00154
          long double c = a, fc = fa, s = b, fs = fb;
00155
          bool mflag = true;
00156
          long double d = 0.0;
00157
00158
          for (int i = 0; i < max_iter; ++i)</pre>
00159
00160
              if (fb != fc && fa != fc)
00161
00162
                   // Inverse quadratic interpolation
                  s = (a * fb * fc) / ((fa - fb) * (fa - fc)) +
(b * fa * fc) / ((fb - fa) * (fb - fc)) +
00163
00164
                       (c * fa * fb) / ((fc - fa) * (fc - fb));
00165
00166
00167
              else
00168
              {
00169
                   // Secant method
                  s = b - fb * (b - a) / (fb - fa);
00170
00171
00172
00173
               // Conditions to accept s
              bool condition1 = (s < (3 * a + b) / 4.0L || s > b);
bool condition2 = (mflag && fabs(s - b) >= fabs(b - c) / 2.0L);
bool condition3 = (!mflag && fabs(s - b) >= fabs(c - d) / 2.0L);
00174
00175
00176
              bool condition4 = (mflag && fabs(b - c) < tol);</pre>
00177
00178
              bool condition5 = (!mflag && fabs(c - d) < tol);
00179
00180
              if (condition1 || condition2 || condition3 || condition4 || condition5)
00181
              {
00182
                  // Bisection method
00183
                  s = (a + b) / 2.0L;
                  mflag = true;
00184
00185
00186
              else
00187
              {
                  mflag = false;
00188
00189
00190
00191
              long double fs_new = f(s);
              00192
00193
00194
00195
              iterations.push_back(oss.str());
00196
00197
              d = c;
```

```
00198
               c = b;
00199
              fc = fb;
00200
               if (fa * fs_new < 0)</pre>
00201
00202
               {
                   b = s;
00203
                  fb = fs_new;
00205
00206
               else
00207
               {
00208
                   a = s;
00209
                  fa = fs_new;
00210
              }
00211
00212
               if (fabs(fa) < fabs(fb))</pre>
00213
                   std::swap(a, b);
00214
00215
                   std::swap(fa, fb);
00216
00217
00218
               if (fabs(b - a) < tol)
00219
                   break;
00220
          }
00221
00222
          return b;
00223 }
```

5.5.1.3 hybrid_method()

```
long double hybrid_method (
              long double a,
              long double b,
              long double tol,
              int max_iter,
              std::vector< std::string > & iterations,
              int decimal_places)
00077 {
00078
         long double fa = f(a), fb = f(b);
00079
          if (fa * fb >= 0)
08000
         {
00081
             std::cerr « "Hybrid method fails. f(a) and f(b) should have opposite signs.\n";
00082
             return NAN;
00083
00084
00085
         long double c = a;
         for (int i = 0; i < max_iter; ++i)</pre>
00086
00087
         {
00088
             c = (a + b) / 2.0L;
00089
             long double fc = f(c);
             std::ostringstream oss;
oss « "Step " « i + 1 « ": [" « std::fixed « std::setprecision(decimal_places) « a « ", " « b
00090
00091
     « "]";
             iterations.push_back(oss.str());
if ((b - a) / 2.0L < tol)</pre>
00092
00093
00094
                 break;
00095
00096
             long double fpc = f_prime(c);
             if (fpc != 0.0)
00097
00098
00099
                 long double d = c - fc / fpc;
00100
                 if (d > a && d < b)
00101
00102
                     long double fd = f(d);
                     std::ostringstream oss_newton;
00103
                     00104
00105
                     iterations.push_back(oss_newton.str());
00106
00107
                     if (fabs(d - c) < tol)
                          return d;
00108
                     if (fa * fd < 0)
00109
00110
                         b = d;
00111
00112
                         fb = fd;
00113
                     else
00114
00115
                         a = d;
00116
00117
                         fa = fd;
00118
00119
                     continue;
```

```
00120
                  }
00121
00122
              // Fallback to bisection
00123
00124
              if (fa * fc < 0)</pre>
00125
00126
                  b = c;
00127
                  fb = fc;
00128
00129
              else
00130
              {
00131
                  a = c;
00132
                  fa = fc;
00133
00134
00135
          return c;
00136 }
```

5.5.1.4 newton_raphson()

```
long double newton_raphson (
              long double x0,
              long double tol,
              int max_iter,
               std::vector< std::string > & iterations,
              int decimal_places)
00052 {
00053
          long double x1;
00054
          for (int i = 0; i < max_iter; ++i)</pre>
00055
00056
              long double fx0 = f(x0);
              long double fpx0 = f_prime(x0);
if (fpx0 == 0.0)
00057
00058
00059
00060
                  std::cerr « "Newton-Raphson method fails. Derivative zero.\n";
00061
00062
              x1 = x0 - fx0 / fpx0;
00063
00064
              std::ostringstream oss;
              oss « "Step " « i + 1 « ": x0 = " « std::fixed « std::setprecision(decimal_places) « x0 « ", x1 = " « x1;
00065
00066
00067
              iterations.push_back(oss.str());
00068
             if (fabs(x1 - x0) < tol)
00069
00070
              x0 = x1;
00071
00072
          return x1;
00073 }
```

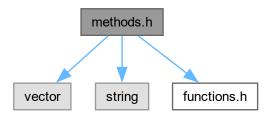
5.5.1.5 ridder_method()

```
long double ridder_method (
             long double a,
              long double b,
              long double tol,
              int max_iter,
              std::vector< std::string > & iterations,
              int decimal_places)
00227 {
00228
          long double fa = f(a), fb = f(b);
          if (fa * fb >= 0)
00229
00230
          {
00231
             std::cerr « "Ridder's method fails. f(a) and f(b) should have opposite signs.\n";
00232
             return NAN;
00233
         }
00234
00235
          for (int i = 0; i < max_iter; ++i)</pre>
00236
00237
             long double c = 0.5L * (a + b);
00238
             long double fc = f(c);
             long double s_sq = fc * fc - fa * fb;
00240
             if (s_sq < 0.0)
```

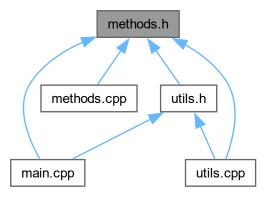
```
{
00242
                    std::cerr « "Ridder's method fails. Square root of negative number.\n";
00243
                     return NAN;
00244
00245
                long double s = sqrt(s_sq);
00246
                if (s == 0.0)
                     return c;
00248
                long double sign = ((fa - fb) < 0) ? -1.0L : 1.0L; long double x = c + (c - a) * fc / s * sign; long double fx = f(x);
00249
00250
00251
00252
                std::ostringstream oss; oss < "Step " < i + 1 < ": [" < std::fixed < std::setprecision(decimal_places) < a < ", " < b
00253
00254
                iterations.push_back(oss.str());
00255
00256
                if (fabs(fx) < tol)</pre>
00257
                     return x;
00259
                if (fc * fx < 0.0)
00260
00261
                    a = c;
                    fa = fc;
b = x;
fb = fx;
00262
00263
00264
00265
00266
                else if (fa \star fx < 0.0)
00267
                    b = x;
00268
                     fb = fx;
00269
00270
00271
                else
00272
00273
                     a = x;
                     fa = fx;
00274
00275
00276
                if (fabs(b - a) < tol)
00278
                     break;
00279
           }
00280
00281
           return 0.5L * (a + b);
00282 }
```

5.6 methods.h File Reference

```
#include <vector>
#include <string>
#include "functions.h"
Include dependency graph for methods.h:
```



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



Classes

struct RootInfo

Functions

- long double bisection (long double a, long double b, long double tol, int max_iter, std::vector< std::string >
 &iterations, int decimal_places)
- long double newton_raphson (long double x0, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double hybrid_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double brent_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)
- long double ridder_method (long double a, long double b, long double tol, int max_iter, std::vector< std::string > &iterations, int decimal_places)

5.6.1 Function Documentation

5.6.1.1 bisection()

```
00023
              return NAN;
00024
00025
          long double c = a;
for (int i = 0; i < max_iter; ++i)</pre>
00026
00027
00028
               c = (a + b) / 2.0L;
00030
               long double fc = f(c);
00031
               std::ostringstream oss;
               oss « "Step " « i + 1 « ": [" « std::fixed « std::setprecision(decimal_places) « a « ", " « b
00032
     « "]";
00033
               iterations.push_back(oss.str());
00034
               if ((b - a) / 2.0L < tol)
00035
                   break;
00036
               if (fa * fc < 0)</pre>
00037
                   b = c;
00038
00039
                   fb = fc;
00040
              }
00041
              else
00042
              {
00043
                   a = c;
00044
                  fa = fc;
00045
00046
00047
          return c;
00048 }
```

5.6.1.2 brent_method()

```
long double brent method (
                  long double a,
                  long double b,
                  long double tol,
                  int max_iter,
                  std::vector< std::string > & iterations,
                  int decimal_places)
00140 {
            long double fa = f(a), fb = f(b);
00141
            if (fa * fb >= 0)
00142
00143
00144
                 std::cerr « "Brent's method fails. f(a) and f(b) should have opposite signs.\n";
00145
00146
            }
00147
00148
            if (fabs(fa) < fabs(fb))</pre>
00149
            {
00150
                 std::swap(a, b);
00151
                 std::swap(fa, fb);
00152
00153
00154
            long double c = a, fc = fa, s = b, fs = fb;
            bool mflag = true;
00155
00156
            long double d = 0.0;
00157
00158
             for (int i = 0; i < max_iter; ++i)</pre>
00159
                 if (fb != fc && fa != fc)
00160
00161
00162
                       // Inverse quadratic interpolation
                      s = (a * fb * fc) / ((fa - fb) * (fa - fc)) +

(b * fa * fc) / ((fb - fa) * (fb - fc)) +

(c * fa * fb) / ((fc - fa) * (fc - fb));
00163
00164
00165
00166
00167
                 else
00168
                 {
00169
                       // Secant method
00170
                      s = b - fb * (b - a) / (fb - fa);
00171
00172
                 // Conditions to accept s bool condition1 = (s < (3 * a + b) / 4.0L || s > b); bool condition2 = (mflag && fabs(s - b) >= fabs(b - c) / 2.0L); bool condition3 = (!mflag && fabs(s - b) >= fabs(c - d) / 2.0L);
00173
00174
00175
00176
                 bool condition4 = (mflag && fabs(b - c) < tol);
bool condition5 = (!mflag && fabs(c - d) < tol);
00177
00178
00179
00180
                  if (condition1 || condition2 || condition3 || condition4 || condition5)
00181
                 {
00182
                       // Bisection method
```

```
s = (a + b) / 2.0L;
00184
               mflag = true;
00185
            }
00186
            else
00187
            {
00188
                mflag = false;
00189
00190
00191
            long double fs_new = f(s);
00192
            std::ostringstream oss;
            00193
00194
00195
            iterations.push_back(oss.str());
00196
00197
            d = c;
            c = b;
fc = fb;
00198
00199
00200
00201
            if (fa * fs_new < 0)</pre>
00202
            {
00203
                b = s;
00204
                fb = fs_new;
00205
            }
00206
            else
00207
            {
00208
                a = s;
00209
                fa = fs_new;
00210
00211
00212
            if (fabs(fa) < fabs(fb))
00213
            {
00214
                std::swap(a, b);
00215
                std::swap(fa, fb);
00216
00217
            if (fabs(b - a) < tol)
00218
00219
                break;
00220
        }
00221
00222
        return b;
00223 }
```

5.6.1.3 hybrid method()

```
long double hybrid_method (
              long double a,
              long double b,
              long double tol,
              int max_iter,
              std::vector< std::string > & iterations,
              int decimal_places)
00077 {
          long double fa = f(a), fb = f(b);
00078
00079
          if (fa * fb >= 0)
08000
00081
              std::cerr \ll "Hybrid method fails. f(a) and f(b) should have opposite signs. \n";
00082
             return NAN;
00083
         }
00084
00085
          long double c = a;
00086
          for (int i = 0; i < max_iter; ++i)</pre>
00087
00088
              c = (a + b) / 2.0L;
00089
             long double fc = f(c);
00090
              std::ostringstream oss;
             oss « "Step " « i + 1 « ": [" « std::fixed « std::setprecision(decimal_places) « a « ", " « b
00091
      « "]";
00092
              iterations.push_back(oss.str());
00093
             if ((b - a) / 2.0L < tol)
                  break:
00094
00095
              long double fpc = f_prime(c);
00096
00097
              if (fpc != 0.0)
00098
00099
                  long double d = c - fc / fpc;
00100
                  if (d > a && d < b)
00101
                  {
00102
                      long double fd = f(d);
00103
                     std::ostringstream oss_newton;
00104
                     oss_newton « "Newton Step: c = " « std::fixed « std::setprecision(decimal_places) « c
```

```
00105
                                 « ", d = " « d;
00106
                       iterations.push_back(oss_newton.str());
00107
                       if (fabs(d - c) < tol)
                          return d;
00108
00109
                       if (fa * fd < 0)
00110
00111
                           b = d;
00112
                           fb = fd;
00113
00114
                       else
00115
00116
                           a = d;
00117
                           fa = fd;
00118
00119
                       continue;
00120
                  }
              }
00121
00122
00123
              // Fallback to bisection
00124
              if (fa * fc < 0)</pre>
00125
              {
                  b = c;
00126
                  fb = fc;
00127
00128
00129
              else
00130
              {
00131
                  a = c;
00132
                  fa = fc;
00133
00134
00135
          return c:
00136 }
```

5.6.1.4 newton_raphson()

```
long double newton_raphson (
               long double x0,
               long double tol,
               int max_iter,
               std::vector< std::string > & iterations,
               int decimal_places)
00052 {
00053
          long double x1;
for (int i = 0; i < max_iter; ++i)</pre>
00054
00055
00056
               long double fx0 = f(x0);
00057
               long double fpx0 = f_prime(x0);
               if (fpx0 == 0.0)
00058
00059
              {
00060
                   std::cerr « "Newton-Raphson method fails. Derivative zero.\n";
00061
                   return NAN;
00062
               x1 = x0 - fx0 / fpx0;
00063
              std::ostringstream oss;
oss « "Step " « i + 1 « ": x0 = " « std::fixed « std::setprecision(decimal_places) « x0 « ", x1 = " « x1;
00064
00065
00066
00067
               iterations.push_back(oss.str());
00068
               if (fabs(x1 - x0) < tol)
               break;
x0 = x1;
00069
00070
00071
00072
          return x1;
00073 }
```

5.6.1.5 ridder_method()

```
long double ridder_method (
          long double a,
          long double b,
          long double tol,
          int max_iter,
          std::vector< std::string > & iterations,
          int decimal_places)
```

5.7 methods.h

```
00227 {
00228
           long double fa = f(a), fb = f(b);
00229
          if (fa * fb >= 0)
00230
          {
               \verb|std::cerr "Ridder's method fails. f(a)| and f(b)| should have opposite signs. \\ \verb|\n";|
00231
00232
               return NAN;
00233
          }
00234
00235
          for (int i = 0; i < max_iter; ++i)</pre>
00236
               long double c = 0.5L * (a + b);
00237
               long double fc = f(c);
00238
00239
               long double s_sq = fc * fc - fa * fb;
00240
               if (s_sq < 0.0)
00241
               {
00242
                   std::cerr « "Ridder's method fails. Square root of negative number.\n";
00243
                   return NAN;
00244
00245
               long double s = sqrt(s_sq);
00246
               if (s == 0.0)
00247
                   return c;
00248
               long double sign = ((fa - fb) < 0) ? -1.0L : 1.0L; long double x = c + (c - a) * fc / s * sign; long double fx = f(x);
00249
00250
00251
00252
               std::ostringstream oss;
00253
               oss < "Step" < i + 1 < ": [" < std::fixed < std::setprecision(decimal_places) < a < ", " < b
      « "]";
00254
               iterations.push_back(oss.str());
00255
00256
               if (fabs(fx) < tol)
00257
                   return x;
00258
00259
               if (fc * fx < 0.0)
00260
                   a = c;
00261
00262
                   fa = fc;
00263
                   b = x;
00264
                   fb = fx;
00265
00266
               else if (fa * fx < 0.0)
00267
               {
                   b = x;
00268
00269
                   fb = fx;
00270
00271
               else
00272
              {
                   a = x;
00273
00274
                   fa = fx;
00275
              }
00276
00277
               if (fabs(b - a) < tol)
00278
                   break;
00279
          }
00280
00281
          return 0.5L * (a + b);
00282 }
```

5.7 methods.h

Go to the documentation of this file.

```
00001 /*
00002 @Author: Gilbert Young
00003 @Time: 2024/09/19 01:47
00004 @File_name: methods.h
00005 @IDE: VSCode
00006 @Formatter: Clang-Format
00007 @Description: Declaration of various root-finding methods.
00008 */
00009
00010 #ifndef METHODS_H
00011 #define METHODS_H
00012
00013 #include <vector>
00014 #include <string>
00015 #include "functions.h"
00016
00017 struct RootInfo
00018 {
          long double {\tt root;} // Root value
00019
                                // Number of iterations
00020
          int iterations;
```

```
int decimal_places; // Number of decimal places to display
00023
00024 // Bisection Method
00025 long double bisection(long double a, long double b, long double tol, int max_iter,
      std::vector<std::string> &iterations, int decimal_places);
00027 // Newton-Raphson Method
00028 long double newton_raphson(long double x0, long double tol, int max_iter, std::vector<std::string>
      &iterations, int decimal_places);
00029
00030 // Hybrid Method (Bisection + Newton-Raphson)
00031 long double hybrid_method(long double a, long double b, long double tol, int max_iter, std::vector<std::string> &iterations, int decimal_places);
00032
00033 // Brent's Method
00034 long double brent_method(long double a, long double b, long double tol, int max_iter,
      std::vector<std::string> &iterations, int decimal_places);
00035
00036 // Ridder's Method
00037 long double ridder_method(long double a, long double b, long double tol, int max_iter,
      std::vector<std::string> &iterations, int decimal_places);
00038
00039 #endif // METHODS_H
```

5.8 plotting.cpp File Reference

```
#include "plotting.h"
#include "functions.h"
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <iomanip>
#include <sstream>
Include dependency graph for plotting.cpp:
```



Functions

• void plot_function (long double x_min, long double x_max, long double y_min, long double y_max, int width, int height, long double label_interval)

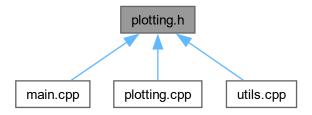
5.8.1 Function Documentation

5.8.1.1 plot_function()

```
long double y_max,
                int width,
                int height,
               long double label_interval)
00022 {
00023
           \verb|std::vector<std::string>| grid(height, std::string(width, ' '));|\\
00024
          int x_axis = -1, y_axis = -1;
00025
00026
           // Determine x-axis position
00027
           if (y_min <= 0 && y_max >= 0)
00028
00029
               x_axis = static_cast < int > (round((0 - y_min) / (y_max - y_min) * (height - 1)));
00030
          }
00031
          // Determine y-axis position
if (x_min <= 0 && x_max >= 0)
00032
00033
00034
00035
               y_{axis} = static_{ast < int > (round((0 - x_min) / (x_max - x_min) * (width - 1)));
00036
          }
00037
00038
           // Plot the function
00039
           for (int i = 0; i < width; ++i)</pre>
00040
               long double x = x_min + i * (x_max - x_min) / (width - 1); long double y = f(x);
00041
00042
               if (y < y_min \mid \mid y > y_max)
00043
00044
                   continue;
00045
               int j = \text{static\_cast} < \text{int} > (\text{round}((y - y_min) / (y_max - y_min) * (\text{height - 1})));
00046
               if (j \ge 0 \&\& j < height)
00047
                   grid[height - 1 - j][i] = ' *';
00048
00049
               }
00050
          }
00051
00052
           // Draw x-axis
00053
           if (x_axis != -1)
00054
00055
               for (int i = 0; i < width; ++i)
00056
               {
00057
                   if (grid[x_axis][i] == ' ')
00058
                       grid[x_axis][i] = '-';
00059
00060
          }
00061
00062
           // Draw y-axis
00063
           if (y_axis != -1)
00064
00065
               for (int i = 0; i < height; ++i)
00066
                   if (grid[i][y_axis] == ' ')
00067
                       grid[i][y_axis] = '|';
00068
00069
               }
00070
          }
00071
00072
           // Draw origin
           if (x_axis != -1 && y_axis != -1)
00073
00074
00075
               grid[x_axis][y_axis] = '+';
00076
00077
          // Print the grid
std::cout « "\nFunction Plot:\n";
00078
00079
           for (const auto &row : grid)
00080
00081
00082
               std::cout « row « '\n';
00083
00084
00085
           // Print x-axis labels
          std::string label_line(width, ' ');
00086
           for (int label = static_cast<int>(ceil(x_min / label_interval)) *
00087
      static_cast<int>(label_interval);
00088
                label <= static_cast<int>(floor(x_max / label_interval)) * static_cast<int>(label_interval);
00089
                label += static_cast<int>(label_interval))
00090
00091
               double relative_pos = (static_cast<double>(label) - x_min) / (x_max - x_min);
               int pos = static_cast<int>(round(relative_pos * (width - 1)));
00092
00093
               std::ostringstream oss_label;
00094
               oss_label « std::fixed « std::setprecision(0) « label;
00095
               std::string label_str = oss_label.str();
00096
               int start_pos = pos - static_cast<int>(label_str.length() / 2);
               if (start_pos < 0)</pre>
00097
00098
                   start_pos = 0;
               if (start_pos + static_cast<int>(label_str.length()) > width)
00099
00100
                    continue;
00101
               for (size_t i = 0; i < label_str.length(); ++i)</pre>
00102
```

5.9 plotting.h File Reference

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



Functions

• void plot_function (long double x_min, long double x_max, long double y_min, long double y_max, int width=60, int height=20, long double label_interval=1.0)

5.9.1 Function Documentation

5.9.1.1 plot_function()

```
void plot_function (
               long double x_min,
               long double x_max,
               long double y_min,
               long double y_max,
               int width = 60,
               int height = 20,
               long double label_interval = 1.0)
00022 {
00023
          std::vector<std::string> grid(height, std::string(width, ' '));
00024
          int x_axis = -1, y_axis = -1;
00025
          // Determine x-axis position
00026
00027
          if (y_min <= 0 && y_max >= 0)
00028
00029
              x_axis = static_cast < int > (round((0 - y_min) / (y_max - y_min) * (height - 1)));
00030
00031
          // Determine y-axis position
if (x_min <= 0 && x_max >= 0)
00032
00033
00034
          {
00035
              y_axis = static_cast < int > (round((0 - x_min) / (x_max - x_min) * (width - 1)));
00036
```

5.10 plotting.h

```
00038
           // Plot the function
00039
           for (int i = 0; i < width; ++i)
00040
               long double x = x_min + i * (x_max - x_min) / (width - 1); long double y = f(x);
00041
00042
               if (y < y_min \mid \mid y > y_max)
00044
                    continue;
00045
               int j = static_cast<int>(round((y - y_min) / (y_max - y_min) * (height - 1)));
00046
               if (j >= 0 && j < height)
00047
00048
                    grid[height - 1 - j][i] = '*';
00049
               }
00050
00051
00052
           // Draw x-axis
           if (x_axis != -1)
00053
00054
           {
               for (int i = 0; i < width; ++i)
00056
               {
                    if (grid[x_axis][i] == ' ')
    grid[x_axis][i] = '-';
00057
00058
00059
               }
00060
           }
00061
00062
           // Draw y-axis
00063
           if (y_axis != -1)
00064
00065
               for (int i = 0; i < height; ++i)
00066
00067
                    if (grid[i][y_axis] == ' ')
00068
                        grid[i][y_axis] = '|';
00069
00070
           }
00071
           // Draw origin
00072
00073
           if (x_axis != -1 && y_axis != -1)
00075
               grid[x_axis][y_axis] = '+';
00076
00077
          // Print the grid
std::cout « "\nFunction Plot:\n";
00078
00079
08000
           for (const auto &row : grid)
00081
00082
               std::cout « row « '\n';
00083
00084
00085
           // Print x-axis labels
           std::string label_line(width, ' ');
00086
           for (int label = static_cast<int>(ceil(x_min / label_interval)) *
00087
      static_cast<int>(label_interval);
00088
                 label <= static_cast<int>(floor(x_max / label_interval)) * static_cast<int>(label_interval);
00089
                label += static_cast<int>(label_interval))
00090
00091
               double relative_pos = (static_cast<double>(label) - x_min) / (x_max - x_min);
               int pos = static_cast<int>(round(relative_pos * (width - 1)));
00093
               std::ostringstream oss_label;
00094
               oss_label « std::fixed « std::setprecision(0) « label;
00095
                std::string label_str = oss_label.str();
               int start_pos = pos - static_cast<int>(label_str.length() / 2);
if (start_pos < 0)</pre>
00096
00097
00098
                    start_pos = 0;
00099
                if (start_pos + static_cast<int>(label_str.length()) > width)
00100
00101
               for (size_t i = 0; i < label_str.length(); ++i)</pre>
00102
               {
                    label_line[start_pos + i] = label_str[i];
00103
00104
00105
           std::cout « label_line « std::endl; std::cout « "x range: [" « x_min « ", " « x_max « "]\n"; std::cout « "y range: [" « y_min « ", " « y_max « "]\n\n";
00106
00107
00108
00109 }
```

5.10 plotting.h

Go to the documentation of this file.

```
00001 /*
00002 @Author: Gilbert Young
00003 @Time: 2024/09/19 01:47
```

5.11 utils.cpp File Reference

```
#include "utils.h"
#include "methods.h"
#include "functions.h"
#include <iostream>
#include <cmath>
#include <iomanip>
#include <sstream>
#include <vector>
#include <map>
#include <string>
#include <functional>
#include #include #include #include
```

Include dependency graph for utils.cpp:



Functions

- void run_method (const std::string &method_name, std::function< long double (long double, long double, long double, int, std::vector< std::string > &, int)> method_func, long double a, long double b, long double tol, int max_iter, int decimal_places)
- void run_problem_steps ()
- void compare_all_methods ()
- void get_user_input (long double &a, long double &b, long double &x0, std::string &method_name, long double &tol)
- int calculate_decimal_places (long double tol)
- void run_method_user_selection (const std::string &method_name, std::function< long double(long double, long double, long double, int, std::vector< std::string > &, int)> method_func, long double a, long double b, long double tol, int max_iter)

Variables

std::map< std::string, std::vector< RootInfo >> summary

5.11.1 Function Documentation

5.11.1.1 calculate_decimal_places()

5.11.1.2 compare_all_methods()

```
void compare_all_methods ()
00142 {
          // Define intervals for three roots
00143
00144
          std::vector<std::pair<long double, long double» intervals = {</pre>
              {-3.0L, -2.0L}, // Negative root
{0.0L, 1.0L}, // First positive root
{1.0L, 3.0L} // Second positive root
00145
00147
              {1.0L, 3.0L}
00148
          };
00149
         // Define tolerances and maximum iterations
00150
00151
          long double tol = 1e-14L; // 14 decimal places
          int max_iter = 1000;
00152
00153
00154
          // List of methods to compare
00155
         std::vector<std::pair<std::string, std::function<long double(long double, long double, long
     00156
     std::vector<std::string> &iterations, int decimal_places) -> long double
00157
             {
00158
                   return bisection(a, b, tol, max_iter, iterations, decimal_places);
00159
              {"Newton-Raphson Method", [](long double a, long double b, long double tol, int max_iter,
00160
     std::vector<std::string> &iterations, int decimal_places) -> long double
             {
00162
                    // For Newton-Raphson, use the midpoint as the initial guess
00163
                   long double initial_guess = (a + b) / 2.0L;
00164
                   return newton_raphson(initial_guess, tol, max_iter, iterations, decimal_places);
00165
     "Hybrid Method", [](long double a, long double b, long double tol, int max_iter,
std::vector<std::string> &iterations, int decimal_places) -> long double
00166
00167
              {
00168
                   return hybrid_method(a, b, tol, max_iter, iterations, decimal_places);
00169
00170
              {"Brent's Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00171
             {
00172
                   return brent_method(a, b, tol, max_iter, iterations, decimal_places);
00173
00174
              {"Ridder's Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00175
             {
00176
                   return ridder method (a, b, tol, max iter, iterations, decimal places);
00177
               }};
00178
00179
          // Store comparison results
00180
          std::map<std::string, std::vector<RootInfo» comparison_results;</pre>
00181
00182
          // Run each method for each interval
00183
          for (const auto &method : methods)
00184
00185
              for (const auto &interval : intervals)
00186
              {
00187
                  std::vector<std::string> iterations;
     long double root = method.second(interval.first, interval.second, tol, max_iter, iterations, 15); // le-14 -> 15 decimal places
00188
                  RootInfo info{root, static_cast<int>(iterations.size()), 15};
```

```
comparison_results[method.first].emplace_back(info);
00191
              }
00192
           }
00193
           \begin{tabular}{ll} // & Display & comparison & table \\ \end{tabular}
00194
00195
          std::cout « "\n--- Comparison of All Methods (Precision: 1e-14) ---\n\n";
00196
00197
00198
           std::cout « std::left « std::setw(25) « "Method"
00199
                      « std::setw(30) « "Root 1 (-3,-2)"
                      « std::setw(15) « "Iterations"
00200
                      « std::setw(30) « "Root 2 (0,1)"
00201
                      « std::setw(15) « "Iterations"
00202
00203
                      « std::setw(30) « "Root 3 (1,3)"
00204
                      « std::setw(15) « "Iterations" « "\n";
00205
00206
           // Separator
          std::cout « std::string(130, '-') « "\n";
00207
00208
00209
           // Table rows
00210
           for (const auto &method : methods)
00211
               std::cout « std::left « std::setw(25) « method.first;
for (size_t i = 0; i < intervals.size(); ++i)</pre>
00212
00213
00214
00215
                    if (comparison_results[method.first][i].root != comparison_results[method.first][i].root)
00216
00217
                        // Check for NAN
                        std::cout « std::left « std::setw(30) « "N/A"
00218
                                   « std::left « std::setw(15) « "N/A";
00219
00220
                    }
00221
                   else
00222
00223
                        std::cout « std::left « std::setw(30) « std::fixed « std::setprecision(15) «
      \verb|comparison_results[method.first][i].root|\\
00224
                                   « std::left « std::setw(15) «
      comparison_results[method.first][i].iterations;
00225
00226
00227
               std::cout « "\n";
00228
           }
00229
00230
           std::cout « "\nNote: Precision is set to 1e-14, output displays 15 decimal places.\n\n";
00231 }
```

5.11.1.3 get user input()

```
void get_user_input (
                                       long double & a,
                                       long double & b,
                                       long double & x0,
                                       std::string & method_name,
                                       long double & tol)
00235 {
00236
                           // List of available methods
                          std::vector<std::string> available_methods = {"Bisection Method", "Hybrid Method", "Brent Method",
00237
               "Ridder Method", "Newton-Raphson Method", "Problem Steps Mode", "Compare All Methods"};
00238
00239
                           // Display available methods
00240
                          std::cout « "\nAvailable methods:\n";
00241
                           for (size_t i = 0; i < available_methods.size(); ++i)</pre>
00242
                                     std::cout « i + 1 « ". " « available_methods[i] « "\n";
00243
00244
                          }
00245
00246
                          // Prompt user to select a method
00247
                           int method_choice;
00248
                           std::cout « "Select a method (1-" « available_methods.size() « "): ";
00249
                          std::cin >> method_choice;
                          while (method_choice < 1 || method_choice > static_cast<int>(available_methods.size()))
00250
00251
00252
                                     \verb|std::cout| & \verb|"Invalid choice. Please select a method (1-" & available_methods.size() & \verb|""; "; available_methods.size() & |""; available_me
00253
                                    std::cin » method_choice;
00254
00255
                          method_name = available_methods[method_choice - 1];
00256
00257
                          if (method name == "Newton-Raphson Method")
00258
00259
                                     // Prompt user to input initial guess x0
00260
                                     std::cout « "Enter initial guess x0: ";
```

```
00261
             std::cin » x0;
00262
          else if (method_name != "Problem Steps Mode" && method_name != "Compare All Methods")
00263
00264
00265
              // Prompt user to input interval [a, b]
00266
              std::cout « "Enter interval [a, b]:\n";
             std::cout « "a = ";
00267
00268
              std::cin » a;
00269
              std::cout « "b = ";
00270
              std::cin » b;
00271
              while (a >= b)
00272
              {
00273
                  std::cout « "Invalid interval. 'a' should be less than 'b'. Please re-enter:\n";
                  std::cout « "a = ";
00274
00275
                  std::cin » a;
00276
                  std::cout « "b = ";
00277
                  std::cin » b;
00278
             }
00279
         }
00280
          if (method_name != "Problem Steps Mode" && method_name != "Compare All Methods")
00281
00282
00283
              // Prompt user to input desired precision
              std::cout « "Enter desired precision (e.g., 1e-14, up to 1e-16): ";
00284
00285
              std::cin » tol;
              const long double min_tol = 1e-16L;
              const long double max_tol = 1e-4L;
00287
00288
              while (tol < min_tol || tol > max_tol)
00289
             {
00290
                 std::cout « "Precision out of bounds (" « min_tol « " to " « max_tol « "). Please
     re-enter: ";
00291
                  std::cin » tol;
00292
00293
00294 }
```

5.11.1.4 run method()

```
void run method (
                const std::string & method_name,
                \mathsf{std}::\mathsf{function}<\mathsf{long}\;\mathsf{double}(\mathsf{long}\;\mathsf{double},\;\mathsf{long}\;\mathsf{double},\;\mathsf{long}\;\mathsf{double},\;\mathsf{int},\;\mathsf{std}\!\!\leftarrow\!\!
::vector< std::string > &, int)> method_func,
               long double a_{i}
               long double b,
               long double tol,
               int max_iter,
               int decimal_places)
00031 {
00032
           std::vector<std::string> iterations;
00033
           long double root = method_func(a, b, tol, max_iter, iterations, decimal_places);
00034
00035
           RootInfo info{root, static_cast<int>(iterations.size()), decimal_places);
00036
           summary[method_name].emplace_back(info);
00037
00038
           // Display results
00039
           std::cout « "\nMethod: " « method_name « "\n";
00040
           if (method_name == "Newton-Raphson Method")
00041
               std::cout « "Initial guess: x0 = " « std::fixed « std::setprecision(decimal_places) « a «
00042
      "\n";
00043
00044
           else
00045
00046
               std::cout « "Interval: [" « std::fixed « std::setprecision(2) « a « ", " « b « "]\n";
00047
           std::cout « "Root: " « std::fixed « std::setprecision(decimal_places) « root « "\n";
00048
           std::cout « "Iterations:\n";
00049
00050
           for (const auto &iter : iterations)
00051
           {
00052
               std::cout « iter « "\n";
00053
00054
           std::cout « "Iterations Count: " « iterations.size() « "\n";
00055 }
```

5.11.1.5 run_method_user_selection()

```
void run\_method\_user\_selection (
```

5.11.1.6 run_problem_steps()

```
void run_problem_steps ()
00059 {
00060
           // Define intervals for three roots
00061
           std::vector<std::pair<long double, long double» intervals = {</pre>
               {-3.0L, -2.0L}, // Negative root
{0.0L, 1.0L}, // First positive root
{1.0L, 3.0L} // Second positive root
00062
00063
00064
               {1.0L, 3.0L}
00065
00066
00067
           // Vector to store roots found in part (i)
00068
           std::vector<long double> found_roots;
00069
00070
           // Define tolerances and maximum iterations
00071
          long double tol_bisection = 1e-4L; // 4 decimal places
long double tol_newton = 1e-14L; // 14 decimal places
long double tol_hybrid = 1e-14L; // 14 decimal places
00072
00073
00074
           int max_iter = 1000;
00075
00076
           std::cout « "\n--- Problem Steps Execution ---\n";
00077
00078
           // Part (i): Bisection Method to find three roots to 4 decimal places
00079
           std::cout \ll "\nPart (i): Bisection Method to find roots to 4 decimal places \n";
00080
           for (const auto &interval : intervals)
00081
00082
               std::vector<std::string> iterations;
               long double root = bisection(interval.first, interval.second, tol_bisection, max_iter,
00083
      iterations, 4);
00084
               RootInfo info{root, static_cast<int>(iterations.size()), 4};
00085
               summary["Bisection Method"].emplace_back(info);
00086
00087
               // Store the found root
00088
               found_roots.emplace_back(root);
00089
               std::cout « "Root in [" « std::fixed « std::setprecision(2) « interval.first « ", " «
00090
      interval.second « "]: "
               00091
00092
00093
          }
00094
00095
           // Part (ii): Newton-Raphson Method to refine the three roots to 14 decimal places
00096
           std::cout \ll "\nPart (ii): Newton-Raphson Method to refine roots to 14 decimal places \n";
           for (auto &x0 : found_roots)
00097
00098
00099
               std::vector<std::string> iterations;
               long double root = newton_raphson(x0, tol_newton, max_iter, iterations, 14);
00100
00101
               RootInfo info{root, static_cast<int>(iterations.size()), 14};
00102
               summary["Newton-Raphson Method"].emplace_back(info);
00103
00104
               std::cout \leftarrow "Refined root starting from " \leftarrow std::fixed \leftarrow std::setprecision(4) \leftarrow x0 \leftarrow ": "
               00105
00106
00107
          }
00108
           // Part (iii): Hybrid Method to find three roots to 14 decimal places
00109
00110
           \texttt{std}::\texttt{cout} \  \, \texttt{``nPart} \  \, (\texttt{iii}): \  \, \texttt{Hybrid Method to find roots to 14 decimal places} \\ \texttt{``n''};
00111
           for (const auto &interval : intervals)
00112
00113
               std::vector<std::string> iterations;
00114
               long double root = hybrid_method(interval.first, interval.second, tol_hybrid, max_iter,
               RootInfo info{root, static_cast<int>(iterations.size()), 14};
00115
               summary["Hybrid Method"].emplace_back(info);
00116
00117
      \verb| std::cout & "Root in [" & std::fixed & std::setprecision(2) & interval.first & ", " & interval.second & "] (Hybrid): " \\
00118
00119
                          « std::fixed « std::setprecision(14) « root « "\n";
```

5.12 utils.h File Reference 33

```
00120
              std::cout « "Iterations: " « iterations.size() « "\n";
00121
00122
          // Output summary of results for problem steps std::cout « "\n--- Summary of Problem Steps Results ---\n";
00123
00124
           for (const auto &method : summary)
00125
00126
00127
               std::cout « "\n" « method.first « "\n";
00128
               int idx = 1;
00129
               for (const auto &info : method.second)
00130
              {
                   std::cout « " Root " « idx++ « ": " « std::fixed « std::setprecision(info.decimal_places)
00131
      « info.root
00132
                              « " | Iterations: " « info.iterations « "\n";
00133
00134
          }
00135
00136
          // Clear summary for next run
          summary.clear();
00137
00138 }
```

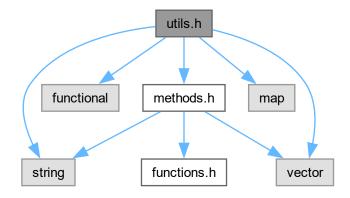
5.11.2 Variable Documentation

5.11.2.1 summary

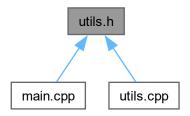
```
std::map<std::string, std::vector<RootInfo> > summary
```

5.12 utils.h File Reference

```
#include <string>
#include <functional>
#include <vector>
#include <map>
#include "methods.h"
Include dependency graph for utils.h:
```



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



Functions

- void run_method (const std::string &method_name, std::function < long double (long double, long double, long double, int, std::vector < std::string > &, int) > method_func, long double a, long double b, long double tol, int max_iter, int decimal_places)
- void run_problem_steps ()
- void compare all methods ()
- void get_user_input (long double &a, long double &b, long double &x0, std::string &method_name, long double &tol)
- int calculate_decimal_places (long double tol)
- void run_method_user_selection (const std::string &method_name, std::function< long double(long double, long double, long double, int, std::vector< std::string > &, int)> method_func, long double a, long double b, long double tol, int max_iter)

Variables

std::map< std::string, std::vector< RootInfo >> summary

5.12.1 Function Documentation

5.12.1.1 calculate decimal places()

5.12 utils.h File Reference 35

5.12.1.2 compare_all_methods()

```
void compare_all_methods ()
00143
           // Define intervals for three roots
00144
          std::vector<std::pair<long double, long double» intervals = {</pre>
              {-3.0L, -2.0L}, // Negative root {0.0L, 1.0L}, // First positive root {1.0L, 3.0L} // Second positive root
00145
00146
00147
              {1.0L, 3.0L}
00148
          };
00149
00150
          \ensuremath{//} Define tolerances and maximum iterations
00151
          long double tol = 1e-14L; // 14 decimal places
          int max_iter = 1000;
00152
00153
00154
          // List of methods to compare
00155
          std::vector<std::pair<std::string, std::function<long double(long double, long double, long
      double, int, std::vector<std::string> &, int) >> methods = {
00156
std::vector<std::string> &iterations, int decimal_places) -> long double 00157
              {"Bisection Method", [](long double a, long double b, long double tol, int max_iter,
               {
00158
                   return bisection(a, b, tol, max_iter, iterations, decimal_places);
00159
00160
              {"Newton-Raphson Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00161
              {
00162
                    // For Newton-Raphson, use the midpoint as the initial guess
00163
                   long double initial_guess = (a + b) / 2.0L;
00164
                   return newton_raphson(initial_guess, tol, max_iter, iterations, decimal_places);
00165
              {"Hybrid Method", [](long double a, long double b, long double tol, int max_iter,
00166
     std::vector<std::string> &iterations, int decimal_places) -> long double
00167
               {
00168
                   return hybrid method(a, b, tol, max iter, iterations, decimal places);
00169
              {"Brent's Method", [](long double a, long double b, long double tol, int max_iter,
     std::vector<std::string> &iterations, int decimal_places) -> long double
00171
              {
00172
                   return brent_method(a, b, tol, max_iter, iterations, decimal_places);
00173
               {"Ridder's Method", [](long double a, long double b, long double tol, int max_iter,
00174
     std::vector<std::string> &iterations, int decimal_places) -> long double
             {
00175
00176
                   return ridder_method(a, b, tol, max_iter, iterations, decimal_places);
00177
               } } ;
00178
00179
          // Store comparison results
00180
          std::map<std::string, std::vector<RootInfo» comparison_results;</pre>
00181
00182
          // Run each method for each interval
00183
          for (const auto &method : methods)
00184
00185
              for (const auto &interval : intervals)
00186
00187
                   std::vector<std::string> iterations;
                   long double root = method.second(interval.first, interval.second, tol, max_iter,
00188
     iterations, 15); // 1e-14 -> 15 decimal places
00189
                  RootInfo info{root, static_cast<int>(iterations.size()), 15};
00190
                  comparison_results[method.first].emplace_back(info);
00191
              }
00192
          }
00193
          // Display comparison table std::cout \ll "\n--- Comparison of All Methods (Precision: le-14) ---\n\n";
00194
00195
00196
00197
          // Table header
00198
          std::cout « std::left « std::setw(25) « "Method"
                     « std::setw(30) « "Root 1 (-3,-2)"
« std::setw(15) « "Iterations"
00199
00200
                     « std::setw(30) « "Root 2 (0,1)"
00201
                     « std::setw(15) « "Iterations"
00202
00203
                     « std::setw(30) « "Root 3 (1,3)"
                     « std::setw(15) « "Iterations" « "\n";
00204
00205
00206
          // Separator
          std::cout « std::string(130, '-') « "\n";
00207
00208
00209
          // Table rows
00210
          for (const auto &method : methods)
00211
00212
               std::cout « std::left « std::setw(25) « method.first;
00213
               for (size_t i = 0; i < intervals.size(); ++i)</pre>
00214
               {
00215
                   if (comparison results[method.first][i].root != comparison results[method.first][i].root)
00216
                   {
00217
                       // Check for NAN
```

```
std::cout « std::left « std::setw(30) « "N/A"
00219
                          « std::left « std::setw(15) « "N/A";
00220
              }
00221
              else
00222
              {
                  std::cout « std::left « std::setw(30) « std::fixed « std::setprecision(15) «
00223
    comparison_results[method.first][i].root
00224
                          « std::left « std::setw(15) «
    comparison_results[method.first][i].iterations;
00225
00226
00227
           std::cout « "\n";
00228
        }
00229
00230
        00231 }
```

5.12.1.3 get user input()

```
void get_user_input (
              long double & a,
               long double & b,
               long double & x0,
               std::string & method_name,
              long double & tol)
00235 {
00236
          // List of available methods
00237
          std::vector<std::string> available_methods = {"Bisection Method", "Hybrid Method", "Brent Method",
      "Ridder Method", "Newton-Raphson Method", "Problem Steps Mode", "Compare All Methods");
00238
00239
          // Display available methods
          std::cout « "\nAvailable methods:\n";
00240
          for (size_t i = 0; i < available_methods.size(); ++i)
00241
00242
00243
              std::cout « i + 1 « ". " « available_methods[i] « "\n";
00244
00245
          // Prompt user to select a method
00246
00247
          int method choice:
00248
          std::cout « "Select a method (1-" « available_methods.size() « "): ";
00249
          std::cin » method_choice;
00250
          while (method_choice < 1 || method_choice > static_cast<int>(available_methods.size()))
00251
              std::cout « "Invalid choice. Please select a method (1-" « available_methods.size() « "): ";
00252
00253
              std::cin » method choice;
00254
00255
          method_name = available_methods[method_choice - 1];
00256
00257
          if (method_name == "Newton-Raphson Method")
00258
00259
              // Prompt user to input initial guess x0
              std::cout « "Enter initial guess x0: ";
00260
00261
              std::cin » x0;
00262
00263
          else if (method_name != "Problem Steps Mode" && method_name != "Compare All Methods")
00264
              // Prompt user to input interval [a, b]
00265
              std::cout « "Enter interval [a, b]:\n";
00266
              std::cout « "a =
00267
00268
              std::cin » a;
00269
              std::cout « "b = ";
00270
              std::cin » b;
00271
              while (a >= b)
00272
              {
00273
                  std::cout « "Invalid interval. 'a' should be less than 'b'. Please re-enter:\n";
00274
                  std::cout « "a = ";
                  std::cin » a;
00275
                  std::cout « "b = ";
00276
00277
                  std::cin » b;
00278
              }
00279
          }
00280
00281
          if (method_name != "Problem Steps Mode" && method_name != "Compare All Methods")
00282
00283
              // Prompt user to input desired precision
              std::cout « "Enter desired precision (e.g., 1e-14, up to 1e-16): ";
00284
00285
              std::cin » tol;
00286
              const long double min_tol = 1e-16L;
              const long double max_tol = 1e-4L;
00287
00288
              while (tol < min_tol || tol > max_tol)
```

5.12 utils.h File Reference 37

5.12.1.4 run_method()

```
void run_method (
                                      const std::string & method_name,
                                      std::function < long double(long double, long double, long double, int, <math>std \leftarrow
::vector< std::string > &, int)> method_func,
                                     long double a,
                                     long double b,
                                     long double tol,
                                      int max_iter,
                                      int decimal_places)
00031 {
00032
                          std::vector<std::string> iterations;
                         long double root = method_func(a, b, tol, max_iter, iterations, decimal_places);
00033
00034
00035
                          RootInfo info{root, static_cast<int>(iterations.size()), decimal_places);
00036
                         summary[method_name].emplace_back(info);
00037
                         // Display results
std::cout « "\nMethod: " « method_name « "\n";
00038
00039
00040
                          if (method_name == "Newton-Raphson Method")
00041
00042
                                    \verb|std::cout| & \verb|"Initial guess: x0 = " & std::fixed & std::setprecision(decimal_places) & a & | & std::setprecision(decimal_places) & | & std::setprecision(decimal
              "\n";
00043
00044
                         else
00045
                         {
00046
                                   std::cout « "Interval: [" « std::fixed « std::setprecision(2) « a « ", " « b « "]\n";
00047
00048
                          std::cout « "Iterations:\n";
00049
00050
                          for (const auto &iter : iterations)
00051
00052
                                   std::cout « iter « "\n";
00053
00054
                          std::cout « "Iterations Count: " « iterations.size() « "\n";
00055 }
```

5.12.1.5 run method user selection()

5.12.1.6 run_problem_steps()

```
std::vector<std::pair<long double, long double» intervals = {</pre>
               {-3.0L, -2.0L}, // Negative root
{0.0L, 1.0L}, // First positive root
{1.0L, 3.0L} // Second positive root
00062
00063
00064
00065
00066
           // Vector to store roots found in part (i)
00067
00068
           std::vector<long double> found_roots;
00069
00070
           // Define tolerances and maximum iterations
           long double tol_bisection = le-4L; // 4 decimal places long double tol_newton = le-14L; // 14 decimal places long double tol_hybrid = le-14L; // 14 decimal places
00071
00072
00073
00074
           int max_iter = 1000;
00075
00076
           std::cout « "\n--- Problem Steps Execution ---\n";
00077
00078
           // Part (i): Bisection Method to find three roots to 4 decimal places
          std::cout « "\nPart (i): Bisection Method to find roots to 4 decimal places\n";
08000
           for (const auto &interval : intervals)
00081
00082
               std::vector<std::string> iterations;
               long double root = bisection(interval.first, interval.second, tol_bisection, max_iter,
00083
      iterations, 4);
00084
               RootInfo info{root, static_cast<int>(iterations.size()), 4};
               summary["Bisection Method"].emplace_back(info);
00085
00086
00087
               // Store the found root
00088
               found_roots.emplace_back(root);
00089
               std::cout « "Root in [" « std::fixed « std::setprecision(2) « interval.first « ", " «
00090
      interval.second « "]: "
00091
                          « std::fixed « std::setprecision(4) « root « "\n";
00092
               std::cout « "Iterations: " « iterations.size() « "\n";
00093
00094
          // Part (ii): Newton-Raphson Method to refine the three roots to 14 decimal places std::cout \ll "\nPart (ii): Newton-Raphson Method to refine roots to 14 decimal places \n";
00095
00097
           for (auto &x0 : found_roots)
00098
00099
               std::vector<std::string> iterations;
00100
               long double root = newton_raphson(x0, tol_newton, max_iter, iterations, 14);
               RootInfo info{root, static_cast<int>(iterations.size()), 14};
00101
00102
               summary["Newton-Raphson Method"].emplace_back(info);
00103
00104
               std::cout « "Refined root starting from " « std::fixed « std::setprecision(4) « x0 « ": "
               00105
00106
          }
00107
00108
00109
           // Part (iii): Hybrid Method to find three roots to 14 decimal places
          std::cout « "\nPart (iii): Hybrid Method to find roots to 14 decimal places\n";
00110
00111
           for (const auto &interval : intervals)
00112
               std::vector<std::string> iterations;
00113
00114
               long double root = hybrid method(interval.first, interval.second, tol hybrid, max iter,
      iterations, 14);
00115
               RootInfo info{root, static_cast<int>(iterations.size()), 14};
00116
               summary["Hybrid Method"].emplace_back(info);
00117
     std::cout « "Root in [" « std::fixed « std::setprecision(2) « interval.first « ", " «
interval.second « "] (Hybrid): "
00118
00119
                         « std::fixed « std::setprecision(14) « root « "\n";
               std::cout « "Iterations: " « iterations.size() « "\n";
00120
00121
          }
00122
          // Output summary of results for problem steps
00123
          std::cout « "\n-- Summary of Problem Steps Results ---\n";
for (const auto &method : summary)
00124
00125
00126
          {
00127
               std::cout « "\nMethod: " « method.first « "\n";
00128
               int idx = 1;
00129
               for (const auto &info : method.second)
00130
               {
                   std::cout « " Root " « idx++ « ": " « std::fixed « std::setprecision(info.decimal_places)
00131
      « info.root
00132
                              « " | Iterations: " « info.iterations « "\n";
00133
00134
          }
00135
00136
          // Clear summary for next run
00137
          summary.clear();
00138 }
```

5.13 utils.h 39

5.12.2 Variable Documentation

5.12.2.1 summary

```
std::map<std::string, std::vector<RootInfo> > summary [extern]
```

5.13 utils.h

Go to the documentation of this file.

```
00002 @Author: Gilbert Young
00003 @Time: 2024/09/19 01:47
00004 @File_name: utils.h
00005 @IDE: VSCode
00006 @Formatter: Clang-Format
00007 @Description: Declarations of utility functions for running methods and handling user input.
00009
00010 #ifndef UTILS_H
00011 #define UTILS_H
00012
00013 #include <string>
00014 #include <functional>
00015 #include <vector>
00016 #include <map>
00017 #include "methods.h"
00018
00019 extern std::map<std::string, std::vector<RootInfo» summary;
00021 // Function to run the method and display results
00022 void run_method(const std::string &method_name,
00023
                    std::function<long double(long double, long double, long double, int,
     00025
                    int decimal_places);
00026
00027 // Function to run the problem steps
00028 void run_problem_steps();
00029
00030 // Function to compare all methods
00031 void compare_all_methods();
00033 // Function to get user input
00034 void get_user_input(long double &a, long double &b, long double &x0, std::string &method_name, long
     double &tol);
00035
00036 // Function to calculate decimal places based on tolerance
00037 int calculate_decimal_places(long double tol);
00039 // Function to run the method and display results (for user-selected methods)
00040 void run_method_user_selection(const std::string &method_name,
00041
                                   std::function<long double(long double, long double, long double, int,</pre>
     std::vector<std::string> &, int)> method_func,
00042
                                   long double a, long double b, long double tol, int max_iter);
00043
00044 #endif // UTILS_H
```

Index

bisection methods.cpp, 14 methods.h, 19 brent_method methods.cpp, 15 methods.h, 20	methods.h, 18 bisection, 19 brent_method, 20 hybrid_method, 21 newton_raphson, 22 ridder_method, 22
calculate_decimal_places utils.cpp, 29 utils.h, 34 compare_all_methods utils.cpp, 29 utils.h, 34	newton_raphson methods.cpp, 17 methods.h, 22 plot_function plotting.cpp, 24 plotting.h, 26
decimal_places RootInfo, 7 f	plotting.rh, 20 plotting.cpp, 24 plot_function, 24 plotting.h, 26 plot_function, 26
functions.cpp, 9 functions.h, 10 f_prime functions.cpp, 9 functions.h, 10 functions.cpp, 9 f, 9 f, 9	ridder_method methods.cpp, 17 methods.h, 22 root RootInfo, 7 Root-Finding Algorithms Solver, 1 RootInfo, 7
f_prime, 9 functions.h, 10 f, 10 f_prime, 10	decimal_places, 7 iterations, 7 root, 7 run_method
get_user_input utils.cpp, 30 utils.h, 36 hybrid_method	utils.cpp, 31 utils.h, 37 run_method_user_selection utils.cpp, 31 utils.h, 37
methods.cpp, 16 methods.h, 21 iterations	run_problem_steps utils.cpp, 32 utils.h, 37
RootInfo, 7 main main.cpp, 12	summary utils.cpp, 33 utils.h, 39
main.cpp, 12 main.cpp, 11 main, 12 methods.cpp, 13 bisection, 14 brent_method, 15 hybrid_method, 16 newton_raphson, 17 ridder_method, 17	utils.cpp, 28 calculate_decimal_places, 29 compare_all_methods, 29 get_user_input, 30 run_method, 31 run_method_user_selection, 31 run_problem_steps, 32 summary, 33

42 INDEX

```
utils.h, 33
calculate_decimal_places, 34
compare_all_methods, 34
get_user_input, 36
run_method, 37
run_method_user_selection, 37
run_problem_steps, 37
summary, 39
```