Appendix C

This section provides the full source code of the program. It starts with RandomWalkCommonHeader.h which defines key global constants and is one of the two important control mechanisms.
This is followed by RandomWalk which contains main().
Next follow the three classes (and their subclasses where applicable) in the order they are used within the program and discussed above. In each case, the header (.h) file is listed first, followed by the implemented (.cpp) file.
The remaining files are listed in alphabetical order.

RandomWalkCommonHeader.h

```
#ifndef RandomWalkCommonHeader_h
    #define RandomWalkCommonHeader_h

    #define CENTRE_INITIAL_STATE 0
    #define CLOCKWISE 0
    #define COUNTER_CLOCKWISE 1
    #define COMPRESSION_FACTOR 1 // Suggested values 1.0 for up to 10,000 steps, 0.1 for
      100,000 steps and 0.05 for 1 million steps
    #define EAST 1
    #define ENABLE_ANALYSIS_MODE() FALSE // It shows how much of Grid.Map has been used.
      Unused Grid.Map means COMPRESSION_FACTOR may be reduce. Aim for a "buffer" of 1.
      Introduces a time overhead!
    #define ENABLE_MULTIPLE_THREADS() TRUE // Splits solutions between available cores, each
      using its own grid map and creating instances of analysis class
    #define ENABLE_LOG_FUNCTION_EXECUTION_TIME() FALSE //:: This can be used to track the
      relative execution time of functions. See GridUtilities.cpp for more information.
      Introduces an overhead!
    #define ENABLE_OUTPUT_ANALYTICS() TRUE //This writes winding angles to analytics class
      and reports mean and std dev to terminal
    #define ENABLE_OUTPUT_WINDING_ANGLES() FALSE // Enabling this functionality writes
      WindingAngles to the display. winding angles are not currently saved.
    #define ENABLE_SAVE_SUCCESSFUL_PATHS() FALSE // We do not need the paths once we have
      the winding angles. A value of FALSE therefore saves memory.
    #define ENABLE_SAVE_WINDING_ANGLES() TRUE // Enabling this function saves winding angle
      to csv file name as specified in RandomWalk.cpp
    #define ENABLE_VERBOSE_MODE() FALSE // Generates considerably more output. It is
      advisable to only use this with small Steps and Solutions to avoid swamoing the
      display with output.
    #define FALSE false
    #define FIRST_VISIT 1
    #define GO_LEFT 3
    #define GO_RIGHT 1
    #define GO_STRAIGHT_ON 0
    #define MAXIMUM_SOLUTION_STEPS 1000000000000
    #define MAXIMUM_STEPS 1000000
    #define NORMAL_TERMINATION 1
    #define NORTH 0
    #define PI 3.141592654
    /* Note that we have to give up one decimal point of precision due to resource
      constraints on the development machine.
       In other words we can multiple radian values between 0 and 2pi by 10 ^ 8 and do
      signed arrithmatic without causing overflow errors.
```

```
   This is at the cost of precision. The cumulative effects should be calculated to see
   if they are significant.
   On a computer with greater memory we would have the option of using larger data
   types. */
#define PRECISION_DECIMAL_POINTS 8
#define SOUTH 2
#define TAB1 "  "
#define TAB2 "      "
#define TAB3 "          "
#define TRUE true
#define UNVISITED 7
#define VERSION "10.0.0 Final Project Code"
#define VISITED_TWICE 6
#define VERY_BIG 100000000
#define WEST 3

/* Below is the information requried for the statistic element of the program,
   The lower and upper bins are defined in winding angles and should be altered
   depending on the ratio of Left to Right.
   These #defines should be removed from RandonWalkCommonHeader and passed on the
   Command Line.*/
#define LOWEST_BIN -1097.5
#define HIGHEST_BIN 1097.5
#define NUMBER_BINS 439

#include <math.h>
#include <time.h>
#include <omp.h>

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

typedef struct Log {
clock_t TotalExecutionTime = 0;
long Invocations = 0;
std::string ID;
std::string FunctionName;
std::string RelativeIndentation = "";
} Log;

unsigned int nextRandomDirectionChange(unsigned int Range);
void xoshiro256ss_init();

inline void fatalError(const std::string& Message){
       std::cout<<"FATAL ERROR: "<<Message<<" Exiting..."<<std::endl;
       exit(EXIT_FAILURE);
    }
#endif
```

RandomWalk.cpp

```
#include "Analytics.h"
#include "Grid.h"
#include "RandomWalkUtilities.h"

#include <cstdlib>
#include <ctime>

int launchThreads(unsigned int& Steps, unsigned int& SolutionsWanted, float Compression,
float ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor,
clock_t& Timer, std::string& Action, std::string& PRightAsText, std::string& PLeftAsText ,
std::vector<Analytics*> &ProgramAnalytics);

int threadMain(unsigned int& Steps, unsigned int& SolutionShare, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t&
Timer, std::string& Action,unsigned short int ThisThread, std::string& PRightAsText,
std::string& PLeftAsText, Analytics* ProgramAnalytics);

int main(int argC, char** argV){
    clock_t Timer = clock();
    // Some information is useful even during VERBOSE_MODE
    std::cout<<std::endl<<"VERSION "<<VERSION<<" BEGINNING AT ("<<(float)(clock()-
      Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;

    float ProbabilityLeft;
    float ProbabilityRight;
    int CallSuccessful;
    std::string Action;
    std::string PLeftAsText;
    std::string PRightAsText;
    unsigned int SolutionsWanted;
    unsigned int Steps;
    /* 100 is multiplied by 10 raised to the power of the Probability Scale Factor to give
      the total Probability Points.
       This enables support for fractional probabilities for straight on, left and right.
       For example if we had straight on = 33.1, right = 33.5 and left = 33.4, we could set
      the scale factor to 1.
       This would give 1,000 probability points: 331 (straight on), 335 (right) and 334
      (left) = 1,000*/
    unsigned short int ProbabilityScaleFactor;

    // Some information is useful even during VERBOSE_MODE
    CallSuccessful = processCommandLine(argC, argV, *&Steps, *&SolutionsWanted,
      *&ProbabilityRight, *&ProbabilityLeft, *&ProbabilityScaleFactor, *&Action,
      *&PLeftAsText, *&PRightAsText);

    if(CallSuccessful == FALSE){
        std::cout<<TAB1<<"Problem with processing command line. Quitting..."<<std::endl;
        displayCommandLineInstructions();
        return -1;
        }

    // Seed random number generator for use in initialisation of xoshirostarstar
      implementation.
    srand(time(0));

    // Initialise custom xoshiro256** random number generator
    xoshiro256ss_init();

    // Initialise class for calculating Analytics
```

```
    std::vector<Analytics*> ProgramAnalytics;
    ProgramAnalytics.resize(omp_get_max_threads());

    // Ensure each thread has access to Analytics class, class uses the number of bins and
       bin range defined in RandomWalkCommonHeader.h
    for(unsigned short int i=0; i<omp_get_max_threads(); i++){
        ProgramAnalytics.at(i)=new Analytics(LOWEST_BIN, HIGHEST_BIN, NUMBER_BINS);
        }
        #if ENABLE_MULTIPLE_THREADS()
            launchThreads(Steps, SolutionsWanted, COMPRESSION_FACTOR, ProbabilityRight,
             ProbabilityLeft, ProbabilityScaleFactor, Timer, Action, PRightAsText,
             PLeftAsText, ProgramAnalytics);
        #else
            threadMain(Steps,SolutionsWanted, COMPRESSION_FACTOR, ProbabilityRight,
             ProbabilityLeft, ProbabilityScaleFactor, Timer, Action, 0, PRightAsText,
             PLeftAsText,  ProgramAnalytics.at(0));
        #endif

   return NORMAL_TERMINATION;
   }

// Intermediate function to execute main() across multiple threads.
int launchThreads(unsigned int& Steps, unsigned int& SolutionsWanted, float Compression,
float ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor,
clock_t& Timer, std::string& Action, std::string& PRightAsText, std::string& PLeftAsText,
std::vector<Analytics*> &ProgramAnalytics){
   unsigned int FirstSolution;
   unsigned int SolutionShare;
   unsigned short int ThisThread;
   unsigned short int Threads;

   #pragma omp parallel default (shared) private(ThisThread, Threads, SolutionShare,
FirstSolution)
   {
      ThisThread = omp_get_thread_num();
      Threads = omp_get_num_threads();
      SolutionShare = SolutionsWanted / Threads;
      FirstSolution = ThisThread * SolutionShare;
      if(ThisThread == Threads-1){ SolutionShare = SolutionsWanted - FirstSolution; }

      threadMain(Steps, SolutionShare, Compression, ProbabilityRight, ProbabilityLeft,
       ProbabilityScaleFactor,Timer, Action, ThisThread, PRightAsText, PLeftAsText,
       ProgramAnalytics.at(ThisThread));
   }

   return NORMAL_TERMINATION;
   }

// Thread code split out of main
int threadMain(unsigned int& Steps, unsigned int& SolutionShare, float Compression, float
ProbabilityRight,float ProbabilityLeft, unsigned short int ProbabilityScaleFactor, clock_t&
Timer, std::string& Action,unsigned short int ThisThread, std::string& PRightAsText,
std::string& PLeftAsText ,Analytics* ProgramAnalytics){
   int CallSuccessful;

    // Create and resize grid and its associated vectors of winding angle differences.
    // Note we intentionally pass Steps rather than Steps + 1.
    Grid SolutionGrid(Steps, COMPRESSION_FACTOR, ProbabilityRight, ProbabilityLeft,
       ProbabilityScaleFactor, ThisThread);

    // Some information is useful even during VERBOSE_MODE.
```

```
std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
   INITIALISING GRID AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<")
   seconds."<<std::endl;
CallSuccessful = SolutionGrid.initialise(Timer, Action);

if(CallSuccessful == FALSE) {std::cout<<TAB1<<"Problem with initialisation.
   Quitting..."<<std::endl; return -1;}

unsigned int Solution = 0;
unsigned int SolutionsFound;
unsigned int SolutionsWanted = SolutionShare;

#if ENABLE_SAVE_WINDING_ANGLES() || ENABLE_OUTPUT_WINDING_ANGLES()
    // Create a vector to hold Winding angles.
    std::vector<float> WindingAngles(SolutionsWanted);
#endif

/* Create vector of vectors to hold all solutions.
Float because we shall be recording winding angles. With more available memory would use
   doubles.
Add 1 to capture the centre position.
Since no need for paths once we have the winding angles, there is an option to discard
   them as we are going along.*/
#if ENABLE_SAVE_SUCCESSFUL_PATHS()
    std::vector<std::vector<signed long>> WalkCollection(SolutionsWanted, vector<signed
   long>(Steps+1));
#else
    std::vector<std::vector<signed long>> WalkCollection(1, vector<signed
   long>(Steps+1));
#endif

// Some information is useful even during VERBOSE_MODE.
std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
   LOOKING FOR SOLUTIONS AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<")
   seconds."<<std::endl;
// Repeatedly call the grid findWalk function until the require number of solutions have
   been found.
for(SolutionsFound = 0; SolutionsFound < SolutionsWanted; SolutionsFound++){
    #if ENABLE_VERBOSE_MODE()
        std::cout<<std::endl<<std::endl<<TAB1<<"Looking for Solution
         "<<SolutionsFound+1<<"."<<std::endl;
    #endif

    #if ENABLE_SAVE_SUCCESSFUL_PATHS()
        Solution = SolutionsFound;
    #endif

    WalkCollection.at(Solution) = SolutionGrid.findWalk();

    #if ENABLE_SAVE_WINDING_ANGLES()
        WindingAngles.at(SolutionsFound) = (WalkCollection.at(Solution).at(Steps) /
         pow(10,PRECISION_DECIMAL_POINTS)) * (180 / PI);
    #endif

    #if ENABLE_OUTPUT_ANALYTICS()
        ProgramAnalytics->push((WalkCollection.at(Solution).at(Steps) /
         pow(10,PRECISION_DECIMAL_POINTS)) * (180 / PI));
    #endif

    #if ENABLE_OUTPUT_WINDING_ANGLES()
        std::cout<<std::endl<<"Solution: "<<SolutionsFound+1<<", Winding angle:
         "<<WindingAngles.at(SolutionsFound)<<"."<<std::endl;
```

```
        #endif
        }

    // Some information is useful even during VERBOSE_MODE.
    std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
       COMPLETED AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Solutions found: "<<SolutionsFound<<"."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Returned to the centre: "<<(unsigned int)
       SolutionGrid.getReturnsToCentre()<<" times."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Loops encountered: "<<(unsigned int)
       SolutionGrid.getLoopsEncountered()<<"."<<std::endl;
    std::cout<<std::endl<<TAB1<<"Average loop length: "<<(float)
       SolutionGrid.getAverageLoopLength()<<"."<<std::endl;

    #if ENABLE_ANALYSIS_MODE()
        /* Provide information about grid use coverage.
           The results can be used to determine whether we get away with a smaller grid.*/
        SolutionGrid.analyseResults();

        /* Investigation of behavioural variance with Professor's code
           Investigate random generation of changes of direction.*/
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
             VARIANCE ANALYSIS."<<std::endl;
        unsigned long TotalDirectionEvents =
SolutionGrid.GoStraightOn+SolutionGrid.TurnLeft+SolutionGrid.TurnRight+SolutionGrid.OtherDir
ectionChange;
        std::cout<<"Total Direction Events: "<<TotalDirectionEvents<<"."<<std::endl;
        std::cout<<TAB1<<"Turn Left: "<<SolutionGrid.TurnLeft<<" ("<<(float)
(100*SolutionGrid.TurnLeft)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"StraightOn: "<<SolutionGrid.GoStraightOn<<" ("<<(float)
(100*SolutionGrid.GoStraightOn)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"Turn Right: "<<SolutionGrid.TurnRight<<" ("<<(float)
(100*SolutionGrid.TurnRight)/TotalDirectionEvents<<"%) ."<<std::endl;
        std::cout<<TAB1<<"Other Direction Change: "<<SolutionGrid.OtherDirectionChange<<"
("<<(float)(100*SolutionGrid.OtherDirectionChange)/TotalDirectionEvents<<"%) ."<<std::endl;
    #endif

    #if ENABLE_SAVE_WINDING_ANGLES()
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
       SAVING "<<WindingAngles.size()<<" WINDING ANGLES AT ("<<(float)(clock()-
       Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
        std::string WindingAngleResults = getWindingAngleFileName(Steps, SolutionsWanted,
       PRightAsText, PLeftAsText, ThisThread);

        writeWindingAnglesToStream(WindingAngleResults, WindingAngles);
    #endif


    #if ENABLE_OUTPUT_ANALYTICS()
        // Some information is useful even during VERBOSE_MODE.
        std::cout<<std::endl<<"VERSION "<<VERSION<<" THREAD: "<<SolutionGrid.getThread()<<"
       SAVING HISTOGRAM AT ("<<(float)(clock()-Timer)/CLOCKS_PER_SEC<<")
       seconds."<<std::endl;

        std::string HistogramResults = getHistogramFileName(Steps, SolutionsWanted,
       PRightAsText, PLeftAsText, ThisThread);
        std::string AnalyticResults = getAnalyticFileName(Steps, SolutionsWanted,
       PRightAsText, PLeftAsText, ThisThread);

        ProgramAnalytics->writeHistogramToStream(HistogramResults);
        ProgramAnalytics->outputResults(AnalyticResults, Steps);
```

```
          // Some information is useful even during VERBOSE_MODE.
        std::cout<<std::endl<<"Analytics:"<<" THREAD:
"<<SolutionGrid.getThread()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Mean Winding Angle: "<<ProgramAnalytics-
>calculateMean()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Variance: "<<ProgramAnalytics-
>calculateVariance()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Variance in Radians: "<<ProgramAnalytics-
>calculateRadianVariance()<<std::endl;
        std::cout<<std::endl<<TAB1<<"Winding Angle Kurtosis: "<<(ProgramAnalytics-
>calculateKurtosis()+3)<<std::endl<<std::endl;
    #endif

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        SolutionGrid.displayExecutionTimes();
    #endif

    return NORMAL_TERMINATION;
    }
```

Grid.h

```
#ifndef Grid_h
    #define Grid_h

    #include "Direction.h"
    #include "RandomWalkCommonHeader.h"

    class Grid{
        friend class Direction;

        public:
            Grid(unsigned int Steps, float Factor, float ProbabilityRight, float
             ProbabilityLeft, unsigned short int ProbabilityScaleFactor, unsigned short int
             ThreadNumber);
            std::vector<signed long> findWalk();
            int initialise(clock_t& Timer, std::string Action);

            // {START: ANALYSIS /UTILITY MEMBER VARIABLES AND FUNCTIONS}
            void analyseResults(); // Implemented in GridUtilities.cpp
            void displayExecutionTimes(); // Implemented in GridUtilities.cpp
            float getAverageLoopLength(); // Implemented in GridUtilities.cpp
            unsigned int getLoopsEncountered(); // Implemented in GridUtilities.cpp
            unsigned int getReturnsToCentre(); // Implemented in GridUtilities.cpp
            unsigned short int getThread(); // Implemented in GridUtilities.cpp
            // {END: ANALYSIS MEMBER VARIABLES AND FUNCTIONS}

            #if ENABLE_ANALYSIS_MODE()
                unsigned long GoStraightOn = 0;
                unsigned long OtherDirectionChange = 0;
                unsigned long TurnLeft = 0;
                unsigned long TurnRight = 0;
            #endif

        private:
            Direction* NewHeadings [4][4];
            DirectionError*  RetracingSteps  = new DirectionError(this, 4,0);
            East*  FacingEast  = new East(this, EAST,2);
            North* FacingNorth = new North(this, NORTH,0);
            South* FacingSouth = new South(this, SOUTH,0);
            West*  FacingWest  = new West(this, WEST,2);

            float GridCompression;
            signed long BoundaryTest;
            unsigned int CentreX;
            unsigned int CentreY;
            unsigned int Centre;
            unsigned int GridPoints;
            unsigned int Length;
            unsigned int LoopsEncountered = 0;
            unsigned int PiAsUnsignedInt = floor(PI*pow(10,PRECISION_DECIMAL_POINTS));
            unsigned int ProbabilityPoints = 100;
            unsigned int ReturnsToCentre = 0;
            unsigned int WalkSteps;
            unsigned long AggregateLoopLength = 0;
            unsigned short int FirstStep;
            unsigned short int FunctionsActuallyLogged = 0;
            unsigned short int Thread;
            std::vector<Log> Logger;
            std::vector<unsigned int> GridMap;
            std::vector<unsigned int> WalkPositionsVisited;
            std::vector<unsigned short int> DirectionChoices;
```

```
            std::vector<unsigned short int> GridMapTrace;
            std::vector<signed long> Walk;
            std::vector<std::string>Reporter;

            int initialiseWalk(unsigned short int& GoingInDirection, unsigned short int&
             NowFacing, unsigned int& Position);
            signed int loadAngleChanges();
            void initialiseLogger();
            void reset(unsigned int& CompletedSteps);
            void serialise();
            void setUpProbabilities(float ProbabilityRight, float ProbabilityLeft, unsigned
             short int ProbabilityScaleFactor);

            // {START: ANALYSIS MEMBER VARIABLES AND FUNCTIONS}
            // These are used and implemented in GridUtilities.cpp
            signed int MaxX = 0;
            signed int MaxY = 0;
            signed int MinY = 0;
            signed int MinX = 0;
            unsigned int TestPosition;
            std::vector<unsigned short int> TestDirections;

            std::string getDirectionAsText(unsigned short int Direction);
            void trackGridUse(unsigned int Step, signed int CurrentX, signed int CurrentY);
            void updateCoordinates(signed int& CurrentX, signed int& CurrentY, unsigned int&
             Position);
            // {END: ANALYSIS MEMBER VARIABLES AND FUNCTIONS}
        };
#endif
```

Grid.cpp

```
#include "Grid.h"

/* PUBLIC
   Grid Constructor processes parameters passed to main() and then resizes its member
vectors to the appropriate size.*/
Grid::Grid(unsigned int Steps, float Compression, float ProbabilityRight, float
ProbabilityLeft,
    unsigned short int ProbabilityScaleFactor, unsigned short int ThreadNumber){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for
       initialiseLogger and setUpProbabilities.
        clock_t LocalTimer = clock();
        initialiseLogger();
    #endif

    // First capture constructor parameters
    Thread = ThreadNumber;
    GridCompression = Compression;
    setUpProbabilities(ProbabilityRight, ProbabilityLeft, ProbabilityScaleFactor);

    /* We add 1 to the length because we want to represent the x = 0 and y = 0 axes withing
       the grid.
       We add a further 2 to the length to create a 1 grid point safe zone all around the
       working grid
       The objective is to avoid an out of bounds error when we create the East, North,
       South and West vectors.*/

    Length = (unsigned int) (ceil(2*Steps*GridCompression)+1);
    // Note that WalkSteps is 1 bigger than the Steps in RandomWalk::main(). This is so that
       we can capture the centre point.
    WalkSteps = Steps + 1;

    if((Length % 2) == 0) Length += 1;
    GridPoints = Length*Length;
    Centre = (GridPoints - 1)/2;

    Walk.resize(WalkSteps);
    WalkPositionsVisited.resize(WalkSteps);
    GridMapTrace.resize(GridPoints,UNVISITED);
    Reporter.resize(WalkSteps*2, "");

    CentreX = CentreY = (Length-1)/2;

    BoundaryTest = ((3/2)*PiAsUnsignedInt)+1;

    FacingEast->setStepSize(Length);
    FacingNorth->setStepSize(1);
    FacingSouth->setStepSize(-1);
    FacingWest->setStepSize(-Length);
    RetracingSteps->setStepSize(0);

    /* The first [] indicates the direction the walk went in to get to current point,
        The second [] indicates the the walk is leaving in from the current point */
    NewHeadings[0][0] = FacingNorth; // Facing North, straight on
    NewHeadings[1][1] = FacingEast; // Facing East, straight on
    NewHeadings[2][2] = FacingSouth; // Facing South, straight on
    NewHeadings[3][3] = FacingWest; // Facing West, straight on

    NewHeadings[0][1] = FacingEast; // Facing North, turn right (Clockwise)
```

```
    NewHeadings[1][2] = FacingSouth; // Facing East, turn right (Clockwise)
    NewHeadings[2][3] = FacingWest; // Facing South, turn right (Clockwise)
    NewHeadings[3][0] = FacingNorth; // Facing West, turn right (Clockwise)

    NewHeadings[0][3] = FacingWest; // Facing North, turn left (Counter clockwise)
    NewHeadings[1][0] = FacingNorth; // Facing East, turn left (Counter clockwise)
    NewHeadings[2][1] = FacingEast; // Facing South, turn left (Counter clockwise)
    NewHeadings[3][2] = FacingSouth; // Facing West, turn left (Counter clockwise)

    NewHeadings[0][2] = RetracingSteps; // Facing North, turning 180 degrees
    NewHeadings[1][3] = RetracingSteps; // Facing East, turning 180 degrees
    NewHeadings[2][0] = RetracingSteps; // Facing South, turning 180 degrees
    NewHeadings[3][1] = RetracingSteps; // Facing West, turning 180 degrees

    // Some information is useful even when ENABLE_VERBOSE_MODE() is set to FALSE.
    std::cout<<TAB2<<"Grid Length: "<<Length<<", Grid Points: "<<GridPoints<<", Walk Steps
      including Step 0 (the centre): "<<WalkSteps<<"."<<std::endl;
    std::cout<<TAB2<<"Centre position: "<<Centre<<", Centre coordinates:
      ("<<CentreX<<","<<CentreY<<")."<<std::endl;

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(0).Invocations += 1;
        Logger.at(0).TotalExecutionTime += (clock() - LocalTimer);
    #endif
    }

/* PUBLIC
   This critical function finds solutions and returns them to the calling function (main()).
It does not know how many solutions are required.
   It uses many of Grid's private member functions.*/
std::vector<signed long> Grid::findWalk(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for
initialiseWalk, reset and Direction::takeOneStep.
        clock_t LocalTimer = clock();
    #endif

    bool SolutionFound;
    bool WalkLoopEncountered;
    signed int CurrentX = 0; // Only used for displaying useful information.
    signed int CurrentY = 0; // Only used for displaying useful information.
    unsigned int LoopCounter = 0;
    unsigned int Position = 0;
    unsigned short int GoingInDirection = 0;
    unsigned short int NowFacing =0; // This represents a different grid point to
GoingInDirection. Together they form a path across a grid point.

    Direction* FacingDirection;

    SolutionFound = FALSE;
    while(SolutionFound == FALSE){

        /* Set all Grid points to unvisited (0) apart from the centre (starting) position
(0,0) which should be set to VISITED_TWICE (900).
           Note that we call this even for "Step 0, when the only affect is to set the
centre (starting) position.
           This reset will also be executed if we have to restart following a loop
encountered event.*/
        reset(*&LoopCounter);
        Position = WalkPositionsVisited.at(0) = Centre;
```

```
        if(initialiseWalk(*&GoingInDirection, *&NowFacing, *&Position) == FALSE)
      fatalError("Grid::findWalk. Inititalise Walk failed.");

        FacingDirection = NewHeadings[GoingInDirection][NowFacing];

        #if ENABLE_VERBOSE_MODE()
            updateCoordinates(*&CurrentX, *&CurrentY, *&WalkPositionsVisited.at(0));
            std::string News = "     THREAD: " + to_string(Thread) + ". Step: 0. Centre
             (starting) position:" + to_string(WalkPositionsVisited.at(0));
            News = News + " (" + to_string(CurrentX) + "," + to_string(CurrentY) + ").
             Position status after initialisation: " +
             to_string(GridMapTrace.at(WalkPositionsVisited.at(0))) + ".\n";
            News = News + TAB3 + "Change in winding angle: " + to_string(Walk.at(0)) + ".
             Cumulative winding angle: " + to_string(Walk.at(0)) + ".\n";
            News = News + TAB3 + "Going " + getDirectionAsText(GoingInDirection) + ".";
            Reporter.at(0) = News;
        #endif

        LoopCounter = 1;
        WalkLoopEncountered = FALSE;
        while( (WalkLoopEncountered == FALSE) && (LoopCounter < WalkSteps) ){
            FacingDirection = FacingDirection->takeOneStep(LoopCounter, *&Position,
             Walk.at(LoopCounter-1), *&Walk.at(LoopCounter), *&WalkLoopEncountered);

            // We record the positions visited so that we can reset GridMapTrace after each
             solution is found.
            WalkPositionsVisited.at(LoopCounter) = Position;

            LoopCounter++;
            }

        /* We need to reset the grid following the successful identification of a solution
      or on encountering a loop.*/
        reset(*&LoopCounter);
        if(LoopCounter == WalkSteps) SolutionFound = TRUE;
        if(WalkLoopEncountered == TRUE) SolutionFound = FALSE;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(6).Invocations += 1;
        Logger.at(6).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    #if ENABLE_VERBOSE_MODE()
        for(int NewsItem = 0; NewsItem < Reporter.size(); NewsItem++)
      std::cout<<Reporter.at(NewsItem);
    #endif

    return Walk;
    }

/* PUBLIC
   This critical function determines the winding angle for each grid element.
   */
int Grid::initialise(clock_t& Timer, std::string Action){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for serialise
and loadWindingAngles.
        clock_t LocalTimer = clock();
    #endif

    signed int VectorSize;
```

```
    signed int XCoordinate;
    signed int YCoordinate;
    unsigned int AbsoluteX;
    unsigned int AbsoluteY;

    if((Action == "serialise")|(Action =="RAM")){ // The only difference between serialise
and RAM is that RAM doesnt output to file.
        std::cout<<TAB1<<"Creating angle change data at ("<<(float)(clock()-
      Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
        GridMap.resize(GridPoints);

        for(unsigned int LoopCounter = 0; LoopCounter < GridPoints; LoopCounter++){
            AbsoluteY = LoopCounter % Length;
            AbsoluteX = (LoopCounter - AbsoluteY) / Length;
            XCoordinate = AbsoluteX - CentreX;
            YCoordinate = AbsoluteY - CentreY;

            GridMap.at(LoopCounter) = floor((atan2(YCoordinate, XCoordinate)+PI) * pow(10,
      PRECISION_DECIMAL_POINTS));
            }

        // We want the centre (starting) point to have a value of 0.
        GridMap.at(Centre) = 0;

        if(Action == "serialise"){
            std::cout<<TAB1<<"Serealising grid map angles at ("<<(float)(clock()-
             Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
            serialise();
            }
        else{
            std::cout<<TAB1<<"Grid map angles (into RAM) at ("<<(float)(clock()-
             Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
            }
        }
    else if(Action == "load"){
        std::cout<<TAB1<<"Loading angle change data at ("<<(float)(clock()-
      Timer)/CLOCKS_PER_SEC<<") seconds."<<std::endl;
        VectorSize = loadAngleChanges();
        if(VectorSize != GridPoints){std::cout<<TAB2<<"ERROR: Unexpected vector size
      ("<<VectorSize<<") vs. expected size("<<GridPoints<<")."<<std::endl;

        #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
            Logger.at(3).Invocations += 1;
            Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
        #endif

        return FALSE;}
        }
    else{
        std::cout<<TAB2<<"ERROR: Unrecognised initialisation
      action:"<<Action<<"."<<std::endl;

        #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
            Logger.at(3).Invocations += 1;
            Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
        #endif

        return FALSE;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(3).Invocations += 1;
```

```
        Logger.at(3).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return TRUE;
    }

/* PRIVATE
   We need to determine our initial direction of movement. We also need to ensure that the
cumulative winding angle is 0 at the centre (Step 0).*/
int Grid::initialiseWalk(unsigned short int& GoingInDirection, unsigned short int&
NowFacing, unsigned int& Position){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    NowFacing = NORTH;
    GoingInDirection = NORTH;

    Walk.at(0) = (signed int) (0-GridMap.at(Position+1));

    FirstStep = NowFacing;
    GridMapTrace.at(Centre) = CENTRE_INITIAL_STATE;

  /* We need to ensure that the Direction Class static class variables which track rotation are
reset. We can use the derived class chosen for Step 1.*/
    NewHeadings[GoingInDirection][NowFacing]->reset();

    Reporter.clear();
    Reporter.resize((WalkSteps*2));

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(7).Invocations += 1;
        Logger.at(7).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return TRUE;
    }

/* PRIVATE
    Load winding angle changes from file.
    Called from initialise()*/
signed int Grid::loadAngleChanges(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    long FileSize;
    size_t Index = 0;
    std::string FileName = "Grid_ST" + to_string(WalkSteps-1) + "CM" +
to_string(COMPRESSION_FACTOR);

   while(TRUE){
      Index = FileName.find(".", Index);
      if(Index == std::string::npos) break;
      FileName.replace(Index, 1, "dot");
        }

    FileName = FileName + ".Map";

    ifstream GridMapIn(FileName, ios::binary);

    if(!GridMapIn.good()){
```

```
        fatalError("Grid::loadAngleChanges. Attempt to load file " + FileName + " failed.");
        }
    else{
        GridMapIn.seekg(0,ifstream::end);
        FileSize = GridMapIn.tellg();
        GridMapIn.seekg(0, ifstream::beg);
        GridMap.resize(FileSize / sizeof(unsigned int));
        GridMapIn.read((char*)&GridMap.at(0),FileSize);
        GridMapIn.close();
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(5).Invocations += 1;
        Logger.at(5).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return FileSize / sizeof(signed int);
    }

/* PRIVATE
   At the start of each attempt to find a solution we must ensure that each grid position is
set to zero apart from the centre (starting) position.*/
void Grid::reset(unsigned int& StepsCompleted){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    // It is a great deal quicker to reset the grid positions visited during the previous
solution / attempt than reset the entire grid each time.
    for(unsigned int LoopCounter = 0; LoopCounter < StepsCompleted; LoopCounter++){
        GridMapTrace.at(WalkPositionsVisited.at(LoopCounter)) = UNVISITED;
        }

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(8).Invocations += 1;
        Logger.at(8).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }

/* PRIVATE
   Writes all grid angles to file.
   Called from initialise() function when Action set to serialise.*/
void Grid::serialise(){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    size_t Index = 0;
    std::string FileName = "Grid_ST" + to_string(WalkSteps-1) + "CM" +
to_string(COMPRESSION_FACTOR);

   while(TRUE){
      Index = FileName.find(".", Index);
      if(Index == std::string::npos) break;
      FileName.replace(Index, 1, "dot");
        }

    FileName = FileName + ".Map";

    ofstream GridMapOut(FileName, ios::out | ios::binary);
```

```
    GridMapOut.write((const char*)&GridMap.at(0), GridMap.size() * sizeof(unsigned int));
    GridMapOut.close();

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(4).Invocations += 1;
        Logger.at(4).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }

/* PRIVATE
   By creating an array of probability weighted direction outcomes at the outset, we improve
subsequent efficiency during searcing for solutions.
   The scale factor lets us deal with non integer probabilities.*/
void Grid::setUpProbabilities(float ProbabilityRight, float ProbabilityLeft, unsigned short
int ProbabilityScaleFactor){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        clock_t LocalTimer = clock();
    #endif

    // To simplify the code and reduce typing later, we first create and calculate the value
of a probality multiplier.
    unsigned int ProbabilityMultiplier = pow((unsigned short int)10, (unsigned short
      int)ProbabilityScaleFactor);

    // Now we scale ProbabilityPoints according to the ProbabilityMultiplier.
    ProbabilityPoints = ProbabilityPoints * ProbabilityMultiplier;

    // In case more decimal places have been used than are multiplied out by the scale
factor, we use floor the left and right probabilities.
    DirectionChoices.resize(floor(ProbabilityRight*ProbabilityMultiplier), GO_RIGHT);
    DirectionChoices.resize(floor((ProbabilityLeft +
      ProbabilityRight)*ProbabilityMultiplier), GO_LEFT);
    DirectionChoices.resize((ProbabilityPoints), GO_STRAIGHT_ON);

    std::cout<<TAB2<<"Probability Points: "<<ProbabilityPoints<<"."<<std::endl;

    #if ENABLE_VERBOSE_MODE()
        std::cout<<TAB3<<" DirectionChoices.size():
      "<<DirectionChoices.size()<<"."<<std::endl;
        std::cout<<TAB3<<" Direction[0]: "<<DirectionChoices.at(0)<<". ";

        if(ProbabilityRight > 0 && ProbabilityRight < 100){
            // Last Right is before position ProbabilityPoints -1.
            std::cout<<" Direction["<<(ProbabilityRight*ProbabilityMultiplier)-1<<"]:
      "<<DirectionChoices.at((ProbabilityRight*ProbabilityMultiplier)-1)<<". ";
            }

        if(ProbabilityRight > 0 && ProbabilityLeft > 0){
            // First Left is not position 0.
            std::cout<<" Direction["<<(ProbabilityRight*ProbabilityMultiplier)<<"]:
      "<<DirectionChoices.at(ProbabilityRight*ProbabilityMultiplier)<<". ";
        }

        if(ProbabilityLeft > 0 && (ProbabilityRight + ProbabilityLeft) < 100){
            // Last Left is before position ProbabilityPoints -1.
            std::cout<<"
      Direction["<<((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)-1<<"]:
      "<<DirectionChoices.at(((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)-
      1)<<". ";
        }
```

```
        if((ProbabilityRight + ProbabilityLeft) > 0 && (ProbabilityRight + ProbabilityLeft)
< 100){
             // First Straight On is not position 0.
             std::cout<<"
       Direction["<<(ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier<<"]:
       "<<DirectionChoices.at((ProbabilityRight+ProbabilityLeft)*ProbabilityMultiplier)<<".
       ";
             }

        std::cout<<" Direction["<<(ProbabilityPoints-1)<<"]:
       "<<DirectionChoices.at(ProbabilityPoints-1)<<"."<<std::endl;
    #endif

    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        Logger.at(2).Invocations += 1;
        Logger.at(2).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return;
    }
```

Direction.h

```
#ifndef Direction_h
    #define Direction_h

    #include "RandomWalkCommonHeader.h"
    class Grid;

    class Direction{
        public:
            Direction(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
             Adjustment);
            void reset();
            virtual void setStepSize(unsigned int Length);
            virtual Direction* takeOneStep(unsigned int& Step, unsigned int& Position,
             signed long& CurrentWindingAngle, signed long& UpdatedWindingAngle, bool&
             WalkLoopEncountered);

        protected:
            // All (instances of all) subclasses need to share the one single value of the
             following two variables.
            // Required to show if the loop has just left the centre.
            static std::vector<bool> JustLeftCentre;
            /* This array (and associated four consts) is used to track changes of winding
             angle following transits through the centre;
               The first dimension represents CameFromDirection - the direction from which
             the Walk entered the centre.
               The second dimension represents NowFacing / - the direction in which the Walk
             will leave the centre.
               The second dimension represents the DirectionOfRotation - this does not
             change as a result of transit through the centre.
               The appropriate value is used to set CentreTransitChange at the grid point
             and it is used on the following set.
               This grid is used in conjunction with the static member bool JustLeftCentre.
               0 represents North, 1 East, 2 South and 3 West.
               The rows below represent the CameFromDirection dimension.
               The in row braces represent the NowFacing dimension.
               The in brace value pairs represent the rotation dimension. The 0 element
             represents counter clockwise. The 1 element represents clockwise.
               */
            const static signed int Ninety = floor(PI*pow(10,PRECISION_DECIMAL_POINTS))/2;
            const static signed int OneEighty = floor(PI*pow(10,PRECISION_DECIMAL_POINTS));
            const static signed int TwoSeventy =
             floor(PI*pow(10,PRECISION_DECIMAL_POINTS))*(3/2);
            const static signed int ThreeSixty =
             floor(PI*pow(10,PRECISION_DECIMAL_POINTS))*2;
            static signed int CentreTransitChanges[4][4][2];
            static std::vector<signed int> PartialRotationEnteringTransit;
            static std::vector<signed int> TransitChange;

            Grid* SolutionGrid;
            signed int StepSize;
            unsigned short int Adjustment;
            unsigned short int Choice;
            unsigned short int CameFromDirection;
            unsigned short int GoingInDirection;
            unsigned short int NowFacing;
        };

    class DirectionError : public Direction{
        public:
```

```
            DirectionError(Grid* ThisGrid, unsigned short int CompassPoint , unsigned short
             int Adjustment);
            Direction* takeOneStep(unsigned int& Step, unsigned int& Position, signed long&
             CurrentWindingAngle, signed long& UpdatedWindingAngle, bool&
             WalkLoopEncountered);
        };

    class East : public Direction{
        public:
            East(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
             Adjustment);
        };

    class North : public Direction{
        public:
            North(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
             Adjustment);
        };

    class South : public Direction{
        public:
            South(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
             Adjustment);
        };

    class West : public Direction{
        public:
            West(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
             Adjustment);
        };
#endif
```

Direction.cpp

```
#include "Direction.h"
#include "Grid.h"

// Define static member variables
std::vector<bool> Direction::JustLeftCentre;
signed int Direction::CentreTransitChanges[4][4][2] = { { {-ThreeSixty,ThreeSixty},{-
Ninety,-Ninety},{-OneEighty,OneEighty},{Ninety,Ninety} },
                                                        { {Ninety,Ninety},{-
ThreeSixty,ThreeSixty},{-Ninety,-Ninety},{-OneEighty,OneEighty} },
                                                        { {-OneEighty,OneEighty},
{Ninety,Ninety},{-ThreeSixty,ThreeSixty},{-Ninety,-Ninety} },
                                                        { {-Ninety,-Ninety},{-
OneEighty,OneEighty},{Ninety,Ninety},{-ThreeSixty,ThreeSixty} } };
std::vector<signed int> Direction::PartialRotationEnteringTransit;
std::vector<signed int> Direction::TransitChange;

/* PUBLIC
   Direction Constructor. By setting its member variables equal to passed parameters, its
subclasses get their differemt behaviours.*/
Direction::Direction(Grid* ThisGrid, unsigned short int CompassPoint, unsigned short int
Adjustment){
    GoingInDirection = CompassPoint;
    CameFromDirection = (GoingInDirection + 2) % 4;
    SolutionGrid = ThisGrid;
    this->Adjustment=Adjustment;

    if(JustLeftCentre.size()!=omp_get_max_threads())
JustLeftCentre.resize(omp_get_max_threads(), FALSE);
    if(PartialRotationEnteringTransit.size()!=omp_get_max_threads())
PartialRotationEnteringTransit.resize(omp_get_max_threads(), 0);
    if(TransitChange.size()!=omp_get_max_threads())
TransitChange.resize(omp_get_max_threads(), 0);
    }

/* PUBLIC
   Derived class constructors simply invoke the base class (Direction) constructor.*/
DirectionError::DirectionError(Grid* ThisGrid, unsigned short int CompassPoint,unsigned
short int Adjustment) : Direction(ThisGrid, CompassPoint, Adjustment){}
East::East(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
North::North(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
South::South(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}
West::West(Grid* ThisGrid, unsigned short int CompassPoint,unsigned short int Adjustment):
Direction(ThisGrid, CompassPoint, Adjustment){}

/* PUBLIC
   This function is needed to reset the Direction classes static state variables which are
shared by all sub class instances.*/
void Direction::reset(){
    JustLeftCentre[SolutionGrid->Thread] = FALSE;
    PartialRotationEnteringTransit[SolutionGrid->Thread] = 0;
    TransitChange[SolutionGrid->Thread] = 0;
    }

/* PUBLIC
```

```
   This function is needed because Length isn't known when the subclasses of Direction are
created and the Direction constructor is called.*/
void Direction::setStepSize(unsigned int Length){
    StepSize = Length;
    }

/* PUBLIC
   This key function is not overriden by the derived classes East, North, South and West.
   However, the derived classes have different behaviour due the specific values of their
member attributes.*/
Direction* Direction::takeOneStep(unsigned int& Step, unsigned int& Position, signed long&
CurrentWindingAngle, signed long& UpdatedWindingAngle, bool& WalkLoopEncountered){
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        // When displaying execution time, we will need to break out the times for
initialiseWalk, getNewDirection navigate and reset.
        clock_t LocalTimer = clock();
    #endif

    #if ENABLE_VERBOSE_MODE() || ENABLE_ANALYSIS_MODE()
        std::string News;
        signed int CurrentX = 0; // Only used for displaying useful information.
        signed int CurrentY = 0; // Only used for displaying useful information.
        std::string ArrivalText = ". ERROR: Somehow this text has not been updated.
      Position: ";
    #endif

    signed long ChangeInWindingAngle;

    // 1. Update GridMapTrace.at(Position) with GoingInDirection.
    SolutionGrid->GridMapTrace.at(Position) += GoingInDirection;
    #if ENABLE_VERBOSE_MODE()
        //std::cout<<" Position status on leaving: "<<SolutionGrid-
      >GridMapTrace.at(Position)<<"."<<std::endl;
        News="Postion status on leaving: " + to_string(SolutionGrid-
      >GridMapTrace.at(Position)) + "\n\n";
        SolutionGrid->Reporter.at((2*Step)-1)=News;
    #endif

    // 2. Take Step
    Position += StepSize;

    // 3. Evaluate the transit history of the new position.
    if(Position != SolutionGrid->Centre){
        // We are not back at the centre.
        if(SolutionGrid->GridMapTrace.at(Position) == UNVISITED){
            // 3.a Arriving at this off centre position for the first time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". New position: ";
            #endif

            // 3.a.1. Calculate the change in winding angle.
            // 3.a.1.i. Determine if this is the very first step.
            if(Step == 1){
                ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
                    (signed long)SolutionGrid->GridMap.at(Position-StepSize);
                /* Ensure that future steps know we have not just left the centre.
                   Note that the following line is not really required since reset() sets
                    JustLeftCentre to FALSE,
                   as does the JustLeftCentre.resize() call in the constructor.*/
                JustLeftCentre[SolutionGrid->Thread] = FALSE;
                }
            // 3.a.1.ii. Determine if we have just passed back through the centre.
```

```
    else if (JustLeftCentre[SolutionGrid->Thread] == TRUE){
        ChangeInWindingAngle = TransitChange[SolutionGrid->Thread] +
            PartialRotationEnteringTransit[SolutionGrid->Thread];

        // Ensure that future steps know we have not just left the centre.
        JustLeftCentre[SolutionGrid->Thread] = FALSE;
        }
    // 3.a.1.iii. An ordinary step.
    else{
        ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
            (signed long)SolutionGrid->GridMap.at(Position-StepSize);

        // We need to adjust for any movement across the 0:2*Pi boundary
        if(ChangeInWindingAngle > SolutionGrid->BoundaryTest) ChangeInWindingAngle =
            ChangeInWindingAngle - (signed long)(2*SolutionGrid->PiAsUnsignedInt);
        else if(ChangeInWindingAngle < -SolutionGrid->BoundaryTest)
            ChangeInWindingAngle = ChangeInWindingAngle + (signed long)
            (2*SolutionGrid->PiAsUnsignedInt);
        }

    /* 3.a.2. Use CameFromDirection (equals GoingInDirection looking backwards).
       NOTE: This is plain =, not += . This is different from below.*/
    SolutionGrid->GridMapTrace.at(Position) = CameFromDirection;

    // 3.a.3. Change Direction with random selection.

    #if ENABLE_ANALYSIS_MODE()
        /* Investigation of behavioural variance with Professor's code.
        Investigate random generation of changes of direction.*/
        Choice =
     SolutionGrid→DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
     >ProbabilityPoints));
        switch (Choice){
            case GO_STRAIGHT_ON: SolutionGrid->GoStraightOn += 1; break;
            case GO_RIGHT: SolutionGrid->TurnRight += 1; break;
            case GO_LEFT: SolutionGrid-> TurnLeft += 1;break;
            default: SolutionGrid->OtherDirectionChange += 1;
        }

        if(Choice == GO_STRAIGHT_ON) NowFacing = GoingInDirection;
        else NowFacing = (GoingInDirection + Choice + Adjustment) % 4;
    #else
        Choice = SolutionGrid-
     >DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
     >ProbabilityPoints));
        // The normal case. No analysis. No testing. Just a standard run
        if(Choice == GO_STRAIGHT_ON) NowFacing =GoingInDirection;
        else NowFacing = (GoingInDirection + Choice +Adjustment) % 4;
    #endif

    }
else{
    // 3.b. Arriving at this off centre position for the second time.
    #if ENABLE_VERBOSE_MODE()
        ArrivalText = ". Previously visited off centre position: ";
    #endif

    // 3.b.1. Calculate the change in winding angle. Note we do not need to test for
     STEP == 0
    // 3.b.1.i. Determine if we have just passed back through the centre.
    if (JustLeftCentre[SolutionGrid->Thread] == TRUE){
```

```
                ChangeInWindingAngle = TransitChange[SolutionGrid->Thread] +
                    PartialRotationEnteringTransit[SolutionGrid->Thread];

                // Ensure that future steps know we have not just left the centre.
                JustLeftCentre[SolutionGrid->Thread] = FALSE;
                }
            // 3.b.1.ii. An ordinary step.
            else{
                ChangeInWindingAngle = (signed long)SolutionGrid->GridMap.at(Position) -
             (signed long)SolutionGrid->GridMap.at(Position-StepSize);

                // We need to adjust for any movement across the 0:2*Pi boundary
                if(ChangeInWindingAngle > SolutionGrid->BoundaryTest) ChangeInWindingAngle =
             ChangeInWindingAngle - (signed long)(2*SolutionGrid->PiAsUnsignedInt);
                else if(ChangeInWindingAngle < -SolutionGrid->BoundaryTest)
             ChangeInWindingAngle = ChangeInWindingAngle + (signed long)(2*SolutionGrid-
             >PiAsUnsignedInt);
                }

            /* 3.b.2. Use CameFromDirection (equals GoingInDirection looking backwards).
               NOTE: This is +=, not plain = . This is different from above.*/
            SolutionGrid->GridMapTrace.at(Position) += CameFromDirection;

            // 3.b.3. Change Direction by subtraction from VISITED_TWICE.
            NowFacing = VISITED_TWICE - SolutionGrid->GridMapTrace.at(Position);
            }
        }
    else{
        // 3.c.1. For testing /analysis purposes we keep track of all the times we have
       returned to the Centre for the entire run.
        SolutionGrid->ReturnsToCentre += 1;

        // 3.c.2. Determine the change in winding angle. For the return to centre step, this
       is the partial or fractional loop rotation value.
        ChangeInWindingAngle = -(CurrentWindingAngle) % (2*SolutionGrid->PiAsUnsignedInt);

        /* 3.c.3. Use CameFromDirection (equals GoingInDirection looking backwards).
           NOTE: This is +=, not plain = . This is different from above.*/
        SolutionGrid->GridMapTrace.at(Position) += CameFromDirection;

        // 3.c.4. The normal case. No analysis. No testing. Just a standard run.
        Choice = SolutionGrid→DirectionChoices.at(nextRandomDirectionChange(SolutionGrid-
       >ProbabilityPoints));
        // The normal case. No analysis. No testing. Just a standard run
        if(Choice==GO_STRAIGHT_ON)NowFacing =GoingInDirection;
        else NowFacing = (GoingInDirection + Choice +Adjustment) % 4;

        // 3.c.5. Determine whether we are back for the first time (or the second and last
time).
        if(SolutionGrid->GridMapTrace.at(Position) < VISITED_TWICE && (NowFacing !=
       SolutionGrid->FirstStep)){
            // 3.c.5.i. We are back at the centre for the first time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". Back at Centre (Starting) position for the first time: ";
            #endif

            // 3.c.5.i.2. We need to determine the angle of rotation at the point of
reaching the centre.
            unsigned short int DirectionOfRotation;
            if(CurrentWindingAngle > 0) DirectionOfRotation = COUNTER_CLOCKWISE; else
       DirectionOfRotation = CLOCKWISE;
```

```
            // 3.c.5.i.3. We need to select the change of direction that must be supplied
and store it where it can be obtained next step.
            TransitChange[SolutionGrid->Thread] = CentreTransitChanges[CameFromDirection]
             [NowFacing][DirectionOfRotation];

            // 3.c.5.i.4. We need to store the partial angle where it can be obtained next
             step.
            PartialRotationEnteringTransit[SolutionGrid->Thread] = (signed int)
             -ChangeInWindingAngle;

            // 3.c.5.i.5. Finally, we need to flag for the next step that we have just left
             the centre.
            JustLeftCentre[SolutionGrid->Thread] = TRUE;
            }
        else{
            // 3.c.5.ii. We are back at the centre for the second time.
            #if ENABLE_VERBOSE_MODE()
                ArrivalText = ". Back at Centre (Starting) position for the second time: ";
            #endif

            // 3.c.5.ii.1. In order for both the display of NowFacing and the return line to
             function properly, we must have a valid NowFacing.*/
            NowFacing = CameFromDirection;

            // 3.c.5.ii.2 We must set the LoopEncountered flag = TRUE.
            WalkLoopEncountered = TRUE;

            // 3.c.5.ii.3 Increment LoopsEncountered and, for interest, update the aggregate
             loop length so that we can later calcuate the average.
            SolutionGrid->LoopsEncountered += 1;
            SolutionGrid->AggregateLoopLength += Step;
            }
        }

    // 4. Update Winding Angle
    UpdatedWindingAngle = (signed long) (CurrentWindingAngle + ChangeInWindingAngle);

    // 5. Return New Direction
    #if ENABLE_VERBOSE_MODE()
        SolutionGrid->updateCoordinates(*&CurrentX, *&CurrentY, *&Position);
        News = "     THREAD: " + to_string(SolutionGrid->Thread) + ". Step: " +
       to_string(Step) + ArrivalText + to_string(Position);
        News = News + " (" + to_string(CurrentX) + "," + to_string(CurrentY) + "). Came from
       the " + SolutionGrid->getDirectionAsText(CameFromDirection);
        News = News + ". Position status having arrived: " + to_string(SolutionGrid-
       >GridMapTrace.at(Position)) + ".\n";
        News = News + TAB3 + "Change in winding angle: " + to_string(ChangeInWindingAngle) +
       ". Cumulative winding angle: " + to_string(UpdatedWindingAngle) + " (";
        News = News + to_string((float)(UpdatedWindingAngle /
       (pow(10,PRECISION_DECIMAL_POINTS)) * (180 / PI))) + ".\n";
        if(ArrivalText != ". Back at Centre (Starting) position for the second time: "){
            News = News + TAB3 + "Going " + SolutionGrid->getDirectionAsText(NowFacing) +
"."
            }
        SolutionGrid->Reporter.at(2*Step) = News;
    #endif

    #if ENABLE_ANALYSIS_MODE()
        SolutionGrid->updateCoordinates(*&CurrentX, *&CurrentY, *&Position);
        SolutionGrid->trackGridUse(Step, CurrentX, CurrentY);
    #endif
```

```
    #if ENABLE_LOG_FUNCTION_EXECUTION_TIME() && !ENABLE_MULTIPLE_THREADS()
        SolutionGrid->Logger.at(9).Invocations += 1;
        SolutionGrid->Logger.at(9).TotalExecutionTime += (clock() - LocalTimer);
    #endif

    return SolutionGrid->NewHeadings[GoingInDirection][NowFacing];
    }

/* PUBLIC
   We need to override the base class virtual function to call fatalError in this case. All
other derived classes use the base class virtual function.*/
Direction* DirectionError::takeOneStep(unsigned int& Step, unsigned int& Position, signed
long& CurrentWindingAngle, signed long& UpdatedWindingAngle, bool& WalkLoopEncountered){
    fatalError("DirectionError::takeOneStep. Requested to retrace steps.");
    return this;
    }
```

Analytics.h

```
#ifndef ANALYTICS_H
    #define ANALYTICS_H

    #include "RandomWalkCommonHeader.h"

    #include "stdlib.h"

    #include <iomanip>
    #include <iostream>
    #include <vector>

    class Analytics{
        public:
            Analytics(double Left, double Right, unsigned int Bins);
            const double calculateKurtosis();
            const double calculateMean();
            const double calculateRadianVariance();
            const double calculateSkewness();
            const double calculateStandardDeviation();
            const double calculateVariance();
            const long long returnNumberOfDataValues();
            void clear();
            const void getHistogramResults(std::vector<double>& binBoundaries,
             std::vector<long>& binData);
            void outputResults(std::string AnalyticResults, unsigned int Steps);
            void push(float x);
            const void writeHistogramToStream(std::string HistogramResults);

        private:
            const double LeftBoundary;
            const double RightBoundary;
            const unsigned long NumberOfBins;

            double BinWidth;
            double M1, M2,radian_M2, M3, M4;
            long long NumberOfAngles;
            vector<long> BinData;
        };
#endif
```

Analytics.cpp

```
#include "Analytics.h"

#include <cmath>
#include <iostream>

/* PUBLIC
   Analytics Constructor uses its parameters to resize BinData and set BindWidth.*/
Analytics::Analytics(double Left, double Right, unsigned int Bins) : LeftBoundary(Left),
RightBoundary(Right), NumberOfBins(Bins){
    clear();

    if(Right <= Left) fatalError("Histogram::Histogram(). Called with right <= left.");
    if(Bins == 0) fatalError("Histogram::Histogram(). Called with nBins == 0.");

    BinWidth = (Right-Left) / Bins;

    /* Histogram will contain nBins between left and right.
       There will also be 2 overspill bins, one for data < left, and one for data >=
right.*/
    BinData.resize(Bins+2);
    }

/* PUBLIC
   Calculate Kurtosis of winding angles.*/
const double Analytics::calculateKurtosis(){
    return double ((NumberOfAngles)*M4 / (M2*M2)) - 3.0;
    }

/* PUBLIC
   Calculate Mean of winding angles.*/
const double Analytics::calculateMean(){
    return M1;
    }

/* PUBLIC
   Calculate Variance of winding angles in radians.*/
const double Analytics::calculateRadianVariance(){
    return radian_M2/(NumberOfAngles-1.0);
    }

/* PUBLIC
   Calculate Skewness of winding angles.*/
const double Analytics::calculateSkewness(){
    return sqrt(double(NumberOfAngles)) * M3 / pow(M2, 1.5);
    }

/* PUBLIC
   Calculate Standard Deviation of winding angles.*/
const double Analytics::calculateStandardDeviation(){
    return sqrt( calculateVariance() );
    }

/* PUBLIC
   Calculate Variance of winding angles.*/
const double Analytics::calculateVariance(){
    return M2/(NumberOfAngles-1.0);
    }

/* PUBLIC
   Clear intermediates and number of data values.*/
```

```
void Analytics::clear(){
    NumberOfAngles = 0;
    M1 = M2 = M3 = M4 = radian_M2 = 0.0;

    return;
    }

/* PUBLIC
   Format results in Histogram.*/
const void Analytics::getHistogramResults(vector<double>& binBoundaries, vector<long>&
binData){
    binBoundaries.resize(NumberOfBins + 2);

    for (unsigned int i = 0; i <= NumberOfBins; ++i){
        binBoundaries[i] = LeftBoundary + static_cast<double>(i) * BinWidth;
        }

    binBoundaries[NumberOfBins + 1] = VERY_BIG;

    binData = BinData;    //    Copy the whole vector

    return;
    }

/* PUBLIC
   Output Analytics to Stream.*/
void Analytics::outputResults(std::string AnalyticResults, unsigned int Steps){
    ofstream AnalyticResultsOutput(AnalyticResults);

    std::cout<<TAB1<<AnalyticResults<<std::endl;

       if (!AnalyticResultsOutput){
         std::cout<<"Failed to open the file."<<std::endl;
         }
       else{
         AnalyticResultsOutput << "Excess_Kurtosis" << ", " << "Kurtosis" << ", " <<
"Variance" << ", " << "Radian_Variance" << ", " << "Mean" << ", " << "Steps" << std::endl;
         AnalyticResultsOutput << calculateKurtosis() << ", " << calculateKurtosis()+3 << ",
" << calculateVariance() << ", " << calculateRadianVariance() << ", " << M1 << ", " << Steps
<< std::endl;
         AnalyticResultsOutput.close();
         }

    return;
    }

/* PUBLIC
   Calculate intermediate terms for analytics without storing individual winding angles.*/
void Analytics::push(float x){
    double delta, delta_n, delta_n2, term1, radian_term1;
    long long n1 = NumberOfAngles;
    NumberOfAngles++;
    delta = x - M1;
    delta_n = delta / NumberOfAngles;
    delta_n2 = delta_n * delta_n;
    term1 = delta * delta_n * n1;
    radian_term1 = (delta*(PI/180)) * (delta_n*(PI/180)) * n1;
    M1 += delta_n;
    M4 += term1 * delta_n2 * (NumberOfAngles*NumberOfAngles - 3*NumberOfAngles + 3) + 6 *
       delta_n2 * M2 - 4 * delta_n * M3;
    M3 += term1 * delta_n * (NumberOfAngles - 2) - 3 * delta_n * M2;
    M2 += term1;
```

```
    radian_M2 += radian_term1;

    //Histogram: Note that if x lies on a boundary, then it gets put in the bin to the
       RIGHT.
    if ( x < LeftBoundary ){
        ++BinData[0];
       }
       else if ( x >= RightBoundary ){
        ++BinData[NumberOfBins + 1];
       }
       else{
        const int iBin = static_cast<int>((x - LeftBoundary) / BinWidth) + 1;
        ++BinData[iBin];
        }

    return;
    }

/* PUBLIC
   Return the number of data values pushed to Analytics Class.*/
const long long Analytics::returnNumberOfDataValues(){
    return NumberOfAngles;
    }

/* PUBLIC
   Output Histogram to Stream.*/
const void Analytics::writeHistogramToStream(std::string HistogramResults){
    ofstream HistogramResultsOutput(HistogramResults);

    std::cout<<TAB1<<HistogramResults<<std::endl;

       if(!HistogramResultsOutput){
        std::cout<<"Failed to open the file."<<std::endl;
        }
       else{
           vector<double> binBoundaries;
           vector<long> binData;
           getHistogramResults(binBoundaries, binData);

           vector<double>::const_iterator BoundaryIterator = binBoundaries.begin();
           vector<long>::const_iterator DataIterator = binData.begin();

           for( ; BoundaryIterator != binBoundaries.end(); ++BoundaryIterator, +
              +DataIterator ){
               HistogramResultsOutput << setw(10) << *BoundaryIterator << ", " <<
                  setw(10) << *DataIterator << std::endl;
             }

        HistogramResultsOutput.close();
        }

    return;
    }
```

GridUtilities.cpp

```
#include "Grid.h"

/* PUBLIC
   Function called during design.
   The function is called when #define ENABLE_ANALYSIS_MODE() is set to TRUE.*/
void Grid::analyseResults(){
    std::cout<<std::endl<<"VERSION "<<VERSION<<" ANALYSIS."<<std::endl;
    std::cout<<TAB1<<"A half axis is "<<(int)(Length-1)/2<<" units."<<std::endl;
    std::cout<<TAB1<<"The minimum value of X reached was  "<<MinX<<"."<<std::endl;
    std::cout<<TAB1<<"The maximum value of X reached was  "<<MaxX<<"."<<std::endl;
    std::cout<<TAB1<<"The minimum value of Y reached was  "<<MinY<<"."<<std::endl;
    std::cout<<TAB1<<"The maximum value of Y reached was  "<<MaxY<<"."<<std::endl;
    return;
    }

/* PUBLIC
   Displays the name of each tracked function, the number of times it was invoked and the
total execution time..
   The function is called when #define ENABLE_LOG_FUNCTION_EXECUTION_TIME() is set to TRUE
   and ENABLE_MULTIPLE_THREADS() is set to FALSE.*/
void Grid::displayExecutionTimes(){
    std::cout<<std::endl<<"VERSION "<<VERSION<<" (TRACKED) FUNCTION EXECUTION
      TIMES."<<std::endl;

    for(short int Count = 0; Count < Logger.size(); Count++){
        std::cout<<TAB1<<Logger.at(Count).RelativeIndentation<<Logger.at(Count).ID<<":
      "<<Logger.at(Count).FunctionName<<" was invoked "<<Logger.at(Count).Invocations;
            std::cout<<" times. The total execution time was
      "<<(float)Logger.at(Count).TotalExecutionTime/CLOCKS_PER_SEC<<" seconds."<<std::endl;
        }

    std::cout<<std::endl<<TAB1<<"Note 1: The logging function imposes a huge overhead and
      when enabled consumes the vast majority of programme execution time."<<std::endl;
    std::cout<<TAB1<<"Note 2: 50% of the time in findWalk is oddly unaccounted
      for."<<std::endl;
    std::cout<<TAB1<<"Note 3: The primary value of the statistics is therefore looking at
      relative execution times across functions."<<std::endl;

    return;
    }

/* PRIVATE
   Converts Direction short int representation to text for reporting progress.
   The function is called when #define ENABLE_VERBOSE_MODE() is set to TRUE.*/
std::string Grid::getDirectionAsText(unsigned short int Direction){
    if (Direction == 1) return "East";
    else if (Direction == 2) return "South";
    else if (Direction == 3) return "West";
    else if (Direction == 0) return "North";
    else{ fatalError("Grid::getDirectionAsText. Direction code " + std::to_string(Direction)
+ " provided."); return "FATAL ERROR"; }
    }

/* PUBLIC
   Used on completion of the search when reporting on the results.*/
float Grid::getAverageLoopLength(){
    return (float) AggregateLoopLength / LoopsEncountered;
    }

/* PUBLIC
```

```
   Used on completion of the search when reporting on the results.*/
unsigned int Grid::getLoopsEncountered(){
    return LoopsEncountered;
    }

/* PUBLIC
   Used on completion of the search when reporting on the results.*/
unsigned int Grid::getReturnsToCentre(){
    return ReturnsToCentre;
    }

/* PUBLIC
    Returns the thread number of the Grid */
unsigned short int Grid::getThread(){
    return Thread;
    }

/* PRIVATE
   Sets up the Logger with the function name of all functions that are being logged.
   In other words it does not log all execution time, but only that spent in functions of
      interest.
   1. In order to reduce the overhead of the logging "function" itself, all chosen functions
      do their own logging.
   2. Discipline is required to ensure that a function updates the correct Logger entry;
   2.1. A copy and paste without subsequent update of the code and a new entry in this
      function will result in;
   2.1.1 The execution time for the new function not being logged and,
   2.2.2 The execution time for the new function will be undetectably added to that for the
      copied function.
   3. The logging function imposes a significant overhead and when enabled consumes a
      significant majority of the time executing the programme.
   3.1 50% of the time in findWalk is unaccounted for.
   4. Where both a function and the function it is called from are logged, both logs will
      include the execution time.
   4.1 Use the RelativeIndentation attribute to indicate functions called from another
      logged function and thus dupliacted execution times.
   The function is called when #define ENABLE_LOG_FUNCTION_EXECUTION_TIME() is set to TRUE.
   This function will be diasabled if ENABLE_MULTIPLE_THREADS() set to TRUE as the results
      and then potentially invalid and no more informative.*/
void Grid::initialiseLogger(){
    // No need to test ENABLE_LOG_FUNCTION_EXECUTION_TIME()
    clock_t LocalTimer = clock();

    Logger.resize(++FunctionsActuallyLogged); Logger.at(0).FunctionName = "Grid()
       constructor"; Logger.at(0).ID = "1";
    Logger.resize(++FunctionsActuallyLogged); Logger.at(1).FunctionName =
       "initialiseLogger"; Logger.at(1).ID = "1.2"; Logger.at(1).RelativeIndentation = TAB1;
    Logger.resize(++FunctionsActuallyLogged); Logger.at(2).FunctionName =
       "setUpProbabilities"; Logger.at(2).ID = "1.1"; Logger.at(2).RelativeIndentation =
       TAB1;
    Logger.resize(++FunctionsActuallyLogged); Logger.at(3).FunctionName = "initialise";
       Logger.at(3).ID = "2";
    Logger.resize(++FunctionsActuallyLogged); Logger.at(4).FunctionName = "serialise";
       Logger.at(4).ID = "2.1"; Logger.at(4).RelativeIndentation = TAB1;
    Logger.resize(++FunctionsActuallyLogged); Logger.at(5).FunctionName =
       "loadAngleChanges"; Logger.at(5).ID = "2.2"; Logger.at(5).RelativeIndentation = TAB1;
    Logger.resize(++FunctionsActuallyLogged); Logger.at(6).FunctionName = "findWalk";
       Logger.at(6).ID = "3";
    Logger.resize(++FunctionsActuallyLogged); Logger.at(7).FunctionName = "initialiseWalk";
       Logger.at(7).ID = "3.1"; Logger.at(7).RelativeIndentation = TAB1;
    Logger.resize(++FunctionsActuallyLogged); Logger.at(8).FunctionName = "reset";
       Logger.at(8).ID = "3.2"; Logger.at(8).RelativeIndentation = TAB1;
```

```
    Logger.resize(++FunctionsActuallyLogged); Logger.at(9).FunctionName =
       "Direction::takeOneStep"; Logger.at(9).ID = "3.3"; Logger.at(9).RelativeIndentation =
       TAB1;

    Logger.at(1).Invocations += 1;
    Logger.at(1).TotalExecutionTime += ((clock() - LocalTimer));

    return;
    }

/* PRIVATE
   The function tracks how far the set of solutions spreads out over the grid.
   The information obtained is used to set the COMPRESSION_FACTOR to allow best performance
      and scalability for a given number of steps.
   The function is called when #define ENABLE_ANALYSIS_MODE() is set to TRUE.*/
void Grid::trackGridUse(unsigned int Step, signed int CurrentX, signed int CurrentY){
    if(CurrentY < MinY) MinY = CurrentY;
    else if(CurrentY > MaxY) MaxY = CurrentY;
    else if(CurrentX < MinX) MinX = CurrentX;
    else if(CurrentX > MaxX) MaxX = CurrentX;

    #if ENABLE_VERBOSE_MODE()
        std::cout<<TAB3<<"MinX:"<<MinX<<", MaxX:"<<MaxX<<", MinY:"<<MinY<<",
MaxY:"<<MaxY<<"."<<std::endl;
    #endif

    return;
    }

/* PRIVATE
   The function converts the grid position expressed as a vector position to one in
coordinates centred in the middle of the grid.
   The function is called when either #define ENABLE_VERBOSE_MODE() or #define
ENABLE_ANALYSIS_MODE() is set to TRUE.*/
void Grid::updateCoordinates(signed int& CurrentX, signed int& CurrentY, unsigned int&
Position){
    CurrentY = (Position % Length) - CentreY;
    CurrentX = ((Position - CurrentY) / Length) - CentreX;

    return;
    }
```

RandomWalkUtilities.h

```
#ifndef RandomWalkUtilities_h
    #define RandomWalkUtilities_h

    #include "RandomWalkCommonHeader.h"

    void displayCommandLineInstructions();
    void displayProgrammeParameters(unsigned int& Steps, unsigned int& SolutionsWanted,
       float& ProbabilityRight, float& ProbabilityLeft, unsigned short int&
       ProbabilityScaleFactor, std::string& Action);
    std::string getAnalyticFileName(unsigned int Steps, unsigned int SolutionsWanted,
       std::string &PRightAsText, std::string &PLeftAsText, unsigned short int Thread);
    std::string getHistogramFileName(unsigned int Steps, unsigned int SolutionsWanted,
       std::string &PRightAsText, std::string &PLeftAsText, unsigned short int Thread);
    std::string getWindingAngleFileName(unsigned int Steps, unsigned int SolutionsWanted,
       std::string& PRightAsText, std::string& PLeftAsText, unsigned short int Thread);
    int processCommandLine(int argC, char** argV, unsigned int& Steps, unsigned int&
       SolutionsWanted, float& ProbabilityRight, float& ProbabilityLeft,unsigned short int&
       ProbabilityScaleFactor, std::string& Action, std::string&  PLeftAsText, std::string&
       PRightAsText);
    void writeWindingAnglesToStream(std::string WindingAngleResults, std::vector<float>&
       WindingAngles);
#endif
```

RandomWalkUtilities.cpp

```
#include "RandomWalkUtilities.h"

/* Outputs key information from RandomWalkCommonHeader.h, the command line and derived
variables once the command line has been correctly validated and processed.
    Called from processCommandLine. */
void displayProgrammeParameters(unsigned int& Steps, unsigned int& SolutionsWanted, float&
ProbabilityRight, float& ProbabilityLeft, unsigned short int& ProbabilityScaleFactor,
std::string& Action){
std::cout<<TAB1<<"[From RandomWalkCommonHeader.h]"<<std::endl;
    std::cout<<TAB2<<"Compression Factor: "<<COMPRESSION_FACTOR<<"."<<std::endl;
    std::cout<<TAB3<<"Note: Suggested values 1.0 for upto 1,000 steps, 0.51 for 10,000
steps, 0.015 for 100,000 steps and 0.0053 for 1 million steps."<<std::endl;
    if(ENABLE_ANALYSIS_MODE()) std::cout<<TAB2<<"Analysis Mode: Enabled."<<std::endl; else
std::cout<<TAB2<<"Analysis Mode: Disabled."<<std::endl;
    if(ENABLE_LOG_FUNCTION_EXECUTION_TIME()) std::cout<<TAB2<<"Log Function Execution Time:
Enabled."<<std::endl; else std::cout<<TAB2<<"Log Function Execution Time:
Disabled."<<std::endl;
    if(ENABLE_MULTIPLE_THREADS()) std::cout<<TAB2<<"Multiple Threads: Enabled."<<std::endl;
else std::cout<<TAB2<<"Multiple Threads: Disabled."<<std::endl;

    if(ENABLE_LOG_FUNCTION_EXECUTION_TIME() && ENABLE_MULTIPLE_THREADS()){
        std::cout<<TAB3<<"Both Log Function Execution Time and Multiple Threads enabled.
Disabling Log Function Execution Time..."<<std::endl;
        }

    if(ENABLE_OUTPUT_ANALYTICS()) std::cout<<TAB2<<"Output Analytics: Enabled."<<std::endl;
else std::cout<<TAB2<<"Output Analytics: Disabled."<<std::endl;
    if(ENABLE_OUTPUT_WINDING_ANGLES()) std::cout<<TAB2<<"Output Winding Angles:
Enabled."<<std::endl; else std::cout<<TAB2<<"Output Winding Angles: Disabled."<<std::endl;
    if(ENABLE_SAVE_SUCCESSFUL_PATHS()) std::cout<<TAB2<<"Perform Save Successful Paths:
Enabled."<<std::endl; else std::cout<<TAB2<<"Perform Save Successful Paths:
Disabled."<<std::endl;
    if(ENABLE_SAVE_WINDING_ANGLES()) std::cout<<TAB2<<"Perform Save Winding Angles:
Enabled."<<std::endl; else std::cout<<TAB2<<"Perform Save Winding Angles:
Disabled."<<std::endl;
    if(ENABLE_VERBOSE_MODE()) std::cout<<TAB2<<"Verbose Mode: Enabled."<<std::endl; else
std::cout<<TAB2<<"Verbose Mode: Disabled."<<std::endl;
    std::cout<<TAB1<<"[From the command line]"<<std::endl;
    std::cout<<TAB2<<"Action on initialisation: "<<Action<<"."<<std::endl;
    std::cout<<TAB2<<"Steps: "<<Steps<<", Solutions Wanted:
"<<SolutionsWanted<<"."<<std::endl;
    std::cout<<TAB2<<"Probability Right: "<<ProbabilityRight<<", Probability Left:
"<<ProbabilityLeft<<", Probability Scale Factor: "<<ProbabilityScaleFactor<<"."<<std::endl;
    std::cout<<TAB1<<"[Important derived variables]"<<std::endl;

    return;
    }

/* Outputs instructions on how to run the programme.
    It is called from main() if the current instance was started with incorrect command line
parameters.*/
void displayCommandLineInstructions(){
    std::cout<<std::endl<<"VERSION "<<VERSION<<" COMMAND LINE INSTRUCTIONS"<<std::endl;
    std::cout<<TAB1<<"1. Six command line parameters are required. They can be supplied in
any order."<<std::endl;
    std::cout<<TAB1<<"2. Parameter names are case sensitive."<<std::endl;
    std::cout<<TAB1<<"3. Parameters must be specified as ParemeterName=ParameterValue. No
spaces are allowed around the '=' sign."<<std::endl;
    std::cout<<TAB1<<"4. Parameters must be separated by space."<<std::endl;
```

```
    std::cout<<TAB1<<"5. Probability Right (PRight) and ProbabilityLeft (PLeft) can be
specified as decimal numbers or as integers."<<std::endl;
    std::cout<<TAB1<<"6. The ProbabilityScaleFactor (PScaleFactor) must be set to at least
the greater of the number of decimal points used for PRight and PLeft."<<std::endl;
    std::cout<<TAB1<<"7. Steps cannot exceed "<<MAXIMUM_STEPS<<"."<<std::endl;
    std::cout<<TAB1<<"8. The product of Steps*Solutions must not exceed
"<<MAXIMUM_SOLUTION_STEPS<<"."<<std::endl;
    std::cout<<TAB1<<"9. Using a new value of Steps may require a change to
COMPRESSION_FACTOR in RandomWalkCommonHeader.h followed by a recompile. See
RandomWalkCommonHeader.h for guidance."<<std::endl;
    std::cout<<TAB1<<"10. Action may take the value 'serialise', RAM or 'load'."<<std::endl;
    std::cout<<TAB1<<"11. Running the programme with Action=serialise will result in the
creation of file Grid.Map."<<std::endl;
    std::cout<<TAB1<<"12. Running the programme with inconsistent Steps and Grid.Map file
will result in an error."<<std::endl;
    std::cout<<TAB1<<"12. Running the programme with Action=RAM will result not result in or
require the creation of file Grid.Map"<<std::endl;
    std::cout<<TAB1<<"13. Running the programme with inconsistent Steps and Grid.Map file
will result in an error."<<std::endl;
    std::cout<<TAB1<<"14. Running the programme with Action=load and no Grid.Map file
present is an uncontrolled condition and the programme will crash."<<std::endl;
    std::cout<<TAB1<<"15. Action=serialise, Action=RAM and Action=load give the same
results. Programme execution with Action=load will be quicker."<<std::endl;
    std::cout<<TAB1<<"16. Additional functionality is enabled / disabled by parameters
specified in RandomWalkCommonHeader.h. Enabling / disabling functionality in this way
requires recompilation."<<std::endl;
    std::cout<<TAB1<<"17. Example: ~/Development/RandomWalk/Version5/RandomWalkV6.4
Solutions=1000 PRight=33 PScaleFactor=0 Steps=2000 Action=load PLeft=33."<<std::endl;

    return;
    }

// Convert relevant run data into string for Histogram File Name
std::string getHistogramFileName(unsigned int Steps, unsigned int SolutionsWanted,
std::string &PRightAsText, std::string &PLeftAsText, unsigned short int Thread){
    size_t Index = 0;
    std::string HistogramFileName = "Histogram_TH";

    HistogramFileName = HistogramFileName +to_string(Thread)+ "ST"+ to_string(Steps) + "SW"
+ to_string(SolutionsWanted) + PRightAsText + PLeftAsText;

    while(TRUE){
        Index = HistogramFileName.find("=", Index);
        if(Index == std::string::npos) break;
        HistogramFileName.replace(Index, 1, "");
        Index += 1;
        }

    Index = 0;
    while(TRUE){
        Index = HistogramFileName.find(".", Index);
        if(Index == std::string::npos) break;
        HistogramFileName.replace(Index, 1, "dot");
        Index += 3;
        }

    Index = 0;
    while(TRUE){
        Index = HistogramFileName.find("PRight", Index);
        if(Index == std::string::npos) break;
        HistogramFileName.replace(Index, 6, "PR");
```

```
        Index = HistogramFileName.find("PLeft", Index);
        if(Index == std::string::npos) break;
        HistogramFileName.replace(Index, 5, "PL"); break;
        }

    HistogramFileName = HistogramFileName + ".csv";

    return HistogramFileName;
    }

// Convert relevant run data into string for Analytics File Name
std::string getAnalyticFileName(unsigned int Steps, unsigned int SolutionsWanted,
std::string &PRightAsText, std::string &PLeftAsText, unsigned short int Thread){
    size_t Index = 0;
    std::string AnalyticFileName = "Analytics_TH";

    AnalyticFileName = AnalyticFileName +to_string(Thread)+ "ST"+ to_string(Steps) + "SW" +
to_string(SolutionsWanted) + PRightAsText + PLeftAsText;

    while(TRUE){
        Index = AnalyticFileName.find("=", Index);
        if(Index == std::string::npos) break;
        AnalyticFileName.replace(Index, 1, "");
        Index += 1;
        }

    Index = 0;
    while(TRUE){
        Index =AnalyticFileName.find(".", Index);
        if(Index == std::string::npos) break;
        AnalyticFileName.replace(Index, 1, "dot");
        Index += 3;
        }

    Index = 0;
    while(TRUE){
        Index = AnalyticFileName.find("PRight", Index);
        if(Index == std::string::npos) break;
        AnalyticFileName.replace(Index, 6, "PR");

        Index = AnalyticFileName.find("PLeft", Index);
        if(Index == std::string::npos) break;
        AnalyticFileName.replace(Index, 5, "PL"); break;
        }

    AnalyticFileName = AnalyticFileName + ".csv";

    return AnalyticFileName;
    }

// Convert relevant run data into string for Winding Angle File Name
std::string getWindingAngleFileName(unsigned int Steps, unsigned int SolutionsWanted,
std::string &PRightAsText, std::string &PLeftAsText, unsigned short int Thread){
    size_t Index = 0;
    std::string WindingAngleFileName = "WindingAngles_TH";

    WindingAngleFileName = WindingAngleFileName +to_string(Thread)+ "ST"+ to_string(Steps) +
"SW" + to_string(SolutionsWanted) + PRightAsText + PLeftAsText;

    while(TRUE){
        Index = WindingAngleFileName.find("=", Index);
        if(Index == std::string::npos) break;
```

```
        WindingAngleFileName.replace(Index, 1, "");
        Index += 1;
        }

    Index = 0;
    while(TRUE){
        Index = WindingAngleFileName.find(".", Index);
        if(Index == std::string::npos) break;
        WindingAngleFileName.replace(Index, 1, "dot");
        Index += 3;
        }

    Index = 0;
    while(TRUE){
        Index = WindingAngleFileName.find("PRight", Index);
        if(Index == std::string::npos) break;
        WindingAngleFileName.replace(Index, 6, "PR");

        Index = WindingAngleFileName.find("PLeft", Index);
        if(Index == std::string::npos) break;
        WindingAngleFileName.replace(Index, 5, "PL"); break;
        }

    WindingAngleFileName = WindingAngleFileName + ".csv";

    return WindingAngleFileName;
    }

/* Processes the arguments passed to the programme from the command ine.
   It is called from main()*/
int processCommandLine(int argC, char** argV, unsigned int& Steps, unsigned int&
SolutionsWanted, float& ProbabilityRight, float& ProbabilityLeft,
    unsigned short int& ProbabilityScaleFactor, std::string& Action, std::string&
PLeftAsText, std::string& PRightAsText){
    std::string Parameter;
    std::string ParameterValuePair[6][2];
    std::string Text;
    unsigned short int ActionParameterCount = 0;
    unsigned short int Item = 0;
    unsigned short int ParametersFound = 0;
    unsigned short int ParameterNumber;
    unsigned short int ParametersSought = 6;
    unsigned short int PLeftParameterCount = 0;
    unsigned short int PRightParameterCount = 0;
    unsigned short int PScaleFactorParameterCount = 0;
    unsigned short int SolutionsParameterCount = 0;
    unsigned short int StepsParameterCount = 0;

    if(argC != ParametersSought+1){
        std::cout<<TAB1<<"ERROR: "<<ParametersSought<<" command line parameters expected.
"<<argC-1<<" command line parameters identified."<<std::endl;
        return FALSE;
        }

    for(ParameterNumber = 1; ParameterNumber < argC; ParameterNumber++){
        istringstream NextPair(argV[ParameterNumber]);
        Item = 0;

        while(getline(NextPair, Text, '=')){
            ParameterValuePair[ParameterNumber-1][Item] = Text;
            Item++;
            }
```

```
        if(ParameterValuePair[ParameterNumber-1][0] == "Steps"){
            Steps = std::stol(ParameterValuePair[ParameterNumber-1][1]);
            ParametersFound++;
            StepsParameterCount++;
            }
        else if(ParameterValuePair[ParameterNumber-1][0] == "Solutions"){
            SolutionsWanted = std::stol(ParameterValuePair[ParameterNumber-1][1]);
            ParametersFound++;
            SolutionsParameterCount++;
            }
        else if(ParameterValuePair[ParameterNumber-1][0] == "PRight"){
            ProbabilityRight = std::stof(ParameterValuePair[ParameterNumber-1][1]);
            ParametersFound++;
            PRightParameterCount++;
            PRightAsText = argV[ParameterNumber];
            }
        else if(ParameterValuePair[ParameterNumber-1][0] == "PLeft"){
            ProbabilityLeft = std::stof(ParameterValuePair[ParameterNumber-1][1]);
            ParametersFound++;
            PLeftParameterCount++;
            PLeftAsText = argV[ParameterNumber];
            }
        else if(ParameterValuePair[ParameterNumber-1][0] == "PScaleFactor"){
            ProbabilityScaleFactor = std::stoi(ParameterValuePair[ParameterNumber-1][1]);
            ParametersFound++;
            PScaleFactorParameterCount++;
            }
        else if(ParameterValuePair[ParameterNumber-1][0] == "Action"){
            Action = ParameterValuePair[ParameterNumber-1][1];
            ParametersFound++;
            ActionParameterCount++;
            }
        else{
            std::cout<<TAB1<<"ERROR: Unrecognised parameter:
             "<<ParameterValuePair[ParameterNumber-1][0]<<"."<<std::endl;
            return FALSE;
            }
        }

    if(Steps > MAXIMUM_STEPS){
        std::cout<<TAB1<<"ERROR: Steps of "<<Steps<<" exceeds supported upper limit of
       "<<MAXIMUM_STEPS<<"."<<std::endl;
        return FALSE;
        }

    if(Steps*SolutionsWanted > MAXIMUM_SOLUTION_STEPS){
        std::cout<<TAB1<<"ERROR: Solution steps (Steps*SolutionsWanted) of
       "<<Steps*SolutionsWanted<<" exceeds supported upper limit of
       "<<MAXIMUM_SOLUTION_STEPS<<"."<<std::endl;
        return FALSE;
        }

    if(Action != "serialise" && Action != "load" && Action != "RAM" ){
        std::cout<<TAB1<<"ERROR: Invalid value of Action specified:
       "<<Action<<"."<<std::endl;
        return FALSE;
        }

    if(StepsParameterCount > 1 || SolutionsParameterCount > 1 || PRightParameterCount > 1 ||
PLeftParameterCount > 1 || PScaleFactorParameterCount > 1 || ActionParameterCount > 1 ){
```

```
        std::cout<<TAB1<<"ERROR: At least one parameter specified multiple
       times."<<std::endl;
        return FALSE;
        }

    displayProgrammeParameters(Steps, SolutionsWanted, ProbabilityRight, ProbabilityLeft,
       ProbabilityScaleFactor, Action);

    return TRUE;
    }

void writeWindingAnglesToStream(std::string WindingAngleResults, std::vector<float>&
WindingAngles){
    ofstream WindingAnglesOutput(WindingAngleResults);

    std::cout<<TAB1<<WindingAngleResults<<std::endl;

       if (!WindingAnglesOutput){
        std::cout<<"Failed to open the file."<<std::endl;
        }
    else{
        for (unsigned i=0; i<WindingAngles.size(); i++) WindingAnglesOutput<<"\n"<<
       WindingAngles.at(i);
        WindingAnglesOutput.close();
        }

    return;
    }
```

xoshirostarstar.cpp

```
/*  Written in 2018 by David Blackman and Sebastiano Vigna (vigna@acm.org)
To the extent possible under law, the author has dedicated all copyright
and related and neighboring rights to this software to the public domain
worldwide. This software is distributed without any warranty.
See <http://creativecommons.org/publicdomain/zero/1.0/>. */

#include <stdint.h>

/* This is xoshiro256** 1.0, one of our all-purpose, rock-solid
   generators. It has excellent (sub-ns) speed, a state (256 bits) that is
   large enough for any parallel application, and it passes all tests we
   are aware of.
   For generating just floating-point numbers, xoshiro256+ is even faster.
   The state must be seeded so that it is not everywhere zero. If you have
   a 64-bit seed, we suggest to seed a splitmix64 generator and use its
   output to fill s. */

/*  RandomWalk Modifications:
      1. Using variable name xoshiro256ss_state in place of s.
      2. jump() and long_jump() functions removed.
      3. next() renamed nextRandomDirectionChange.
      4. Replace uint64_t with unsigned long.
      5. Replaced int with unsigned  int.
      6. nextRandomDirectionChange() modified to return number between 0 and passed in
      Range-1.
      7. Custom RandomWalk initialisation function added.*/

static inline unsigned long rotl(const unsigned long x, signed short int k){
      return (x << k) | (x >> (64 - k));
    }

static unsigned long xoshiro256ss_state[4]; // Replaces static uint64_t s[4];

unsigned int nextRandomDirectionChange(unsigned int Range){
      const unsigned int result = (rotl(xoshiro256ss_state[1] * 5, 7) * 9) % Range;

      const unsigned long t = xoshiro256ss_state[1] << 17;

      xoshiro256ss_state[2] ^= xoshiro256ss_state[0];
      xoshiro256ss_state[3] ^= xoshiro256ss_state[1];
      xoshiro256ss_state[1] ^= xoshiro256ss_state[2];
      xoshiro256ss_state[0] ^= xoshiro256ss_state[3];

      xoshiro256ss_state[2] ^= t;

      xoshiro256ss_state[3] = rotl(xoshiro256ss_state[3], 45);

      return result;
      }

// Custom RandomWalk initialisation function added.
#include <stdlib.h>
void xoshiro256ss_init(){
      xoshiro256ss_state[0] = rand();
      xoshiro256ss_state[1] = rand();
      xoshiro256ss_state[2] = rand();
      xoshiro256ss_state[3] = rand();

      return;
      }
```