A Simulation of a Liquid-Based Natural Algorithm for Finding the Average of n Integers Using a Cellular Automaton

Dominik Schultes

8. April 2004

Abstract

We use a Cellular Automaton (CA) to simulate a liquid-based natural algorithm for finding the average of n integers. A modeling of the problem in terms of a CA is presented, some properties are specified that have to be fulfilled by the CA in order to be able to solve the problem and, finally, one concrete set of rules is derived. Appendix A contains one test run and Appendix B the source code of a C++ program [Sch04] that implements the described CA.

1 Introduction

In order to compute the average of n numbers, we can use a liquid-based natural algorithm [Aru04]. Each number is represented by a corresponding water level in a cylinder, where each cylinder has the same diameter. When the lower parts of all cylinders are connected, the water is distributed to all cylinders equally because of the equal atmospheric pressure so that the water level of all cylinders represents the average value.

We want to use a Cellular Automaton (CA) [Wol86] to simulate this algorithm. In order to do so, we have to restrict the input to integers as we can only represent discrete values. Let m be the biggest number of the n integers. Then, we can use a CA with n columns and m rows to simulate the algorithm. Each column corresponds to one cylinder resp. to one number of the input. Initially, for each column x all cells $1 \leq y \leq a_x$ are *filled*, where a_x is the x-th number of the input. The remaining cells are *empty*.

The CA should have the following properties:

1. In each step, the number of filled cells does not change because we want to simulate a closed system where nothing is added and nothing is taken away.

2. After a finite amount of steps, the automaton reaches a state that represents the correct result and all the following states represent the correct result as well. If the average of the n integers is an integer k, there is only one state that represents the correct result, namely, the state where the rows from 1 to k are filled completely and all other cells are empty. If the average is a rational number $r, k < r < k + 1$, a correct state consists of the k completely filled rows and exactly j filled cells in the row $k + 1$, where $r = k + j/n, 1 \leq j < n$.

3. After a finite amount of steps, the automaton reaches a final state and stays in this state forever.

The Properties 1 and 2 are absolutely required, while the Property 3 is helpful in order to be able to decide if a correct state is reached: Property 3 guarantees that the algorithm terminates after a finite amount of steps, i.e., the state does not change anymore, and together with Property 2 we know that this final state represents a correct solution.

2 Basic Rules

Basing on the above mentioned general properties, there are several sets of rules that lead to a CA that solves the problem correctly. We want to present one simple set of rules that achieves that goal.

1. If a filled cell is above an empty cell, both cells are swapped.

2. If a filled cell is left above an empty cell, both cells are swapped.

3. If a filled cell is right above an empty cell, both cells are swapped.

The order of these rules corresponds with the arbitrarily chosen priority, i.e., if all three conditions are fulfilled by a filled cell, it changes the place with the empty cell below; if there is a filled cell above another filled cell whose horizontal neighbours are empty, the former filled cell "goes" down to the right (Rule 2).

Unfortunately, these rules can easily cause conflicts, i.e., up to three filled cells want to change places with the same empty cell. This has to be avoided in order to obey Property 1. It is possible to prevent these conflicts in a conventional CA by introducing additional states and by using a bigger neighbourhood so that cells in a square with edges of length five influence the next state of a cell instead of just the cells in a square with edges of length three, where only direct neighbours are considered. However, the description

of such an automaton would get quite complex. Therefore, we prefer to introduce *sub-steps*, which leads to a simple description: Each rule is encapsulated in a sub-step. During one step all sub-steps are executed in the order that corresponds with the priority of the rule. If the state of one cell changes during a sub-step, this cell is specially marked so that it is skipped in the following sub-steps. After all sub-steps have been performed, the markers are removed for the next step. Due to this partition of a step, we do not have to explicitly handle collisions.

3 Advanced Rules

The three rules from Section 2 represent the fact that the water is pressed downwards by the atmospheric pressure, but they are not sufficient. For instance, nothing would happen if the input was $1, 2, 3, 4, 5$. Hence, we need some additional rules so that the water first flows sidewards in order to be able to flow downwards. On principle, a CA makes only local decisions, but here we need a kind of global decision. Let us compare the instances $0, 1, 1, 2, 1, 1, 1$ and $1, 1, 1, 2, 1, 1, 0$. In the former, the 2 should go to the left in order to fill the 0; in the later, the 2 should go to the right. As we cannot make a global decision, we just choose one direction, say right, and the filled cell changes its place with an empty cell to its right as long as possible. Then the direction is inverted so that we can be sure that a "hole", i.e., a possibility to move downwards, is discovered (if there is any). But, we have to remember what we have already seen. Otherwise, the filled cell cannot know if it should move right or if it has already reached the right border so that it has to move left now. Hence, we replace the state *filled* by two states *right* and *left*. However, we need even more states. If a right cell hits something on its right, it turns back and goes to the left. But, we have to distinguish between two cases. The first case is that the right border has been touched or that the obstruction is a filled cell that as already touched the right border. In this case, we know that there is no hole to the right of this point and we store this knowledge by changing to the state *left_trb* for "going left, already touched the right boundary". The second case is that the obstruction is just an arbitrary filled cell. In this case, we only switch to the state *left* as we cannot be sure that there is no hole to the right of the current point. When a left cell reaches the left border, its state switches to *right_tlb*.

Thus, we add the following rules:

4.
   - If a cell that is in the state *right* or *right_tlb* is left to an empty cell, both cells are swapped.
   - If a *right* cell is left to a non-empty cell, it adopts the state of the other cell.
   - If a *right* cell is left to the left boundary, the state of the cell is changed to *left_trb*.
5.
   - If a cell that is in the state *left* or *left_trb* is right to an empty cell, both cells are swapped.
   - If a *left* cell is right to a non-empty cell, it adopts the state of the other cell.
   - If a *left* cell is right to the left boundary, the state of the cell is changed to *right_tlb*.
   - If a *left_trb* cell is right to the left boundary or to a *right_tlb* cell, the state of the cell is changed to *right_tlb*.

The Rules 4 and 5 form a fourth and fifth sub-step. The subdivision of one step into five sub-steps still ensures that Property 1 holds. Furthermore, Property 2 is fulfilled due to the cooperation of the five rules. If exactly k rows have been completely filled, then only the row $k+1$ can contain non-empty cells. If there was a non-empty cell in a row $j > k+1$, it would find a "hole" in the row $k+1$, which exists as the row $k+1$ is the first uncompleted row. Then, the hole would be occupied and the non-empty cell in the row j would disappear. Moreover, Property 3 is fulfilled as well. The final state of each non-empty cell is *right_tlb*, i.e., after a cell has touched the left boundary (or another cell that has touched the left boundary), it moves to the right as long as possible. In the end, k rows are completely filled with *right_tlb* cells, and all non-empty cells in the row $k+1$ are aligned at the right boundary. Then, the state of all cells is fixed and the algorithm terminates with the correct result.

References

[Aru04] J. J. Arulanandham. Introducing natural algorithms. http://www.cs.auckland.ac.nz/∼cristian/umc/Introduction.zip, March 2004.

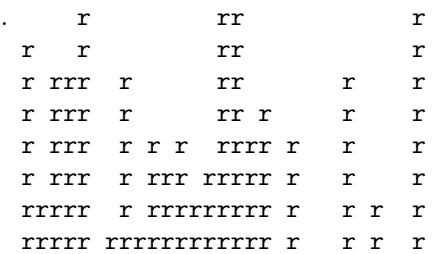
[Sch04] Dominik Schultes. A simulation of a liquid-based natural algorithm for finding the average of n integers using a cellular automaton. http://www-user.rhrk.uni-kl.de/∼dschult/umc/asg3/, April 2004.

[Wol86] S. Wolfram. *Theory and applications of cellular automata*. World Scientific, 1986.

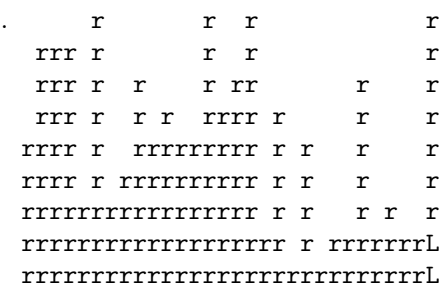
A Test Run

The following test run demonstrates the behaviour of the CA for a randomly chosen example with $n = 30$ and $m = 10$. The states of the cells are represented in the following way: *empty* = whitespace, *right* = `r`, *left* = `l`, *right_tlb* = `R`, *left_trb* = `L`.

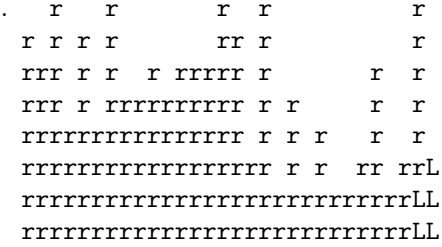
0.



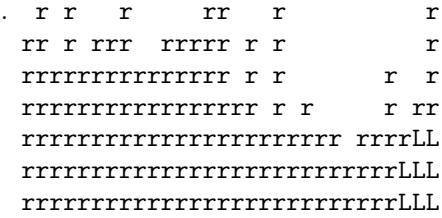
1.



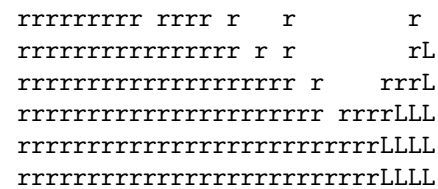
2.



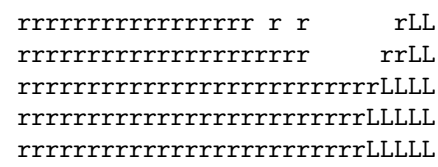
3.



4.

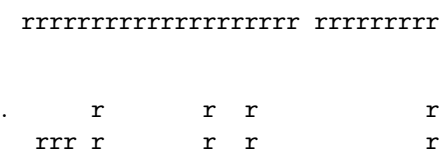


5.



10.

15.

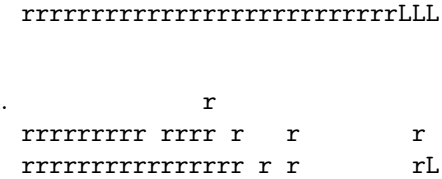


20.

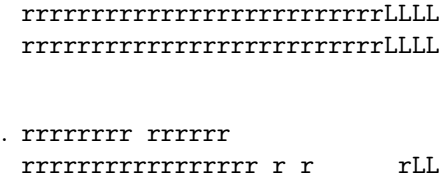
25.

30.

35.



40.



45. r
RRRRRRRRRLLLLLLLLLLLLLLLLLLLL
RRRRRRRRRRLLLLLLLLLLLLLLLLLLL
RRRRRRRRRRRRLLLLLLLLLLLLLLLLL
RRRRRRRRRRRRRRLLLLLLLLLLLLLLL
RRRRRRRRRRRRRRRLLLLLLLLLLLLLL

50. L
RRRRRRRRRRRRRRLLLLLLLLLLLLLLL
RRRRRRRRRRRRRRRLLLLLLLLLLLLLL
RRRRRRRRRRRRRRRRRRLLLLLLLLLLL
RRRRRRRRRRRRRRRRRRRLLLLLLLLLL
RRRRRRRRRRRRRRRRRRRLLLLLLLLLL

60. L
RRRRRRRRRRRRRRRRRRRRRRRRLLLLLL
RRRRRRRRRRRRRRRRRRRRRRRRRLLLL
RRRRRRRRRRRRRRRRRRRRRRRRRRRRL
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR

70. L
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR

80. R
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR

100. R
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR

106. R
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRR

B Source Code

```
/*****************************
 * A Simulation of a Liquid-Based Natural Algorithm for
 * Finding the Average of n Integers Using a Cellular
 * Automaton
 *
 * by Dominik Schultes
 * 8. April 2004
 *
 * Designed for Linux 2.4.19 and the g++ compiler 3.2.
 * At least the method "clrscr" has to be adapted if
 * Windows is used !
 *****************************/


#include <algorithm>
#include <iostream>

using namespace std;


/**
   Clears the screen. This method is platform-dependent !
   It has to be adapted if Windows instead of Linux is used.
*/
void clrscr() {
    system("clear");
}


/**
   This class encapsulates the Cellular Automaton (CA)
   that is used to determine the average of several
   integers.
*/
class AverageCA
{
 private:
    // The possible states of each cell
    static const int OUT_OF_BOUNDS = 0;
    static const int EMPTY = 1;
    static const int RIGHT = 2;
    static const int LEFT = 3;
    static const int RIGHT_TLB = 4;
    static const int LEFT_TRB = 5;

    /**
       This class encapsulates the cells of the automaton.
    */
    class Cells {
     public:
        Cells() {}

        /**
           Create a two-dimensional array of integers,
           where each integer represents the state of
           the corresponding cell.
        */
        Cells(int n, int m) : _n( n ), _m( m ) {
            _cells = new int[n * m];
        }

        ~Cells() {
            delete[] _cells;
        }

        /** Returns the state of the specified cell. */
        int getState(int x, int y) {
            if ((x < 0) || (x >= _n) || (y < 0) || (y >= _m))
                return OUT_OF_BOUNDS;
```

```
        return _cells[ x*_m + y ];
    }

    /** Sets the state of the specified cell. */
    void setState(int x, int y, int newState) {
        _cells[ x*_m + y ] = newState;
    }

 private:
    int _n; // the number of integers
    int _m; // the value of the biggest integer

    // a two-dim. array that represents the cells
    int* _cells;
};

public:
  /** Creates and initializes the Cellular Automaton. */
  AverageCA(int n, int input[])
      : _n( n ), _clocks( 0 ), _changed( 0 ) {
      // determine the maximum integer, compute the sum
      // of all integers, and compute the desired value
      // (the average of all integers) (in order to be
      // able to check the result)
      _max = 0;
      int sum = 0;
      for (int i=0; i<n; i++) {
          _max = max(_max, input[i]);
          sum += input[i];
      }
      _desiredValue = sum / (float)n;

      // create the cells
      _cells = new Cells(n, _max);

      // initialize the states of the cells
      for (int x=0; x<n; x++) {
          for (int y=0; y<input[x]; y++)
              setState(x,y,RIGHT);
          for (int y=input[x]; y<_max; y++)
              setState(x,y,EMPTY);
      }
  }

  ~AverageCA() {
      delete _cells;
  }

  /**
     Returns true, iff no cells changed during the last
     step.
  */
  bool isFinished() {
      return (_changed == 0);
  }

  /**
     Returns the average of all integers, which has been
     computed with conventional means.
  */
  float getDesiredValue() {
      return _desiredValue;
  }

  /**
     Returns the average of all integers, which has been
     computed by the Cellular Automaton.
  */
  float getActualValue() {
      return getNoOfCompletedRows() +
          (getNoOfNonEmptyCellsOnFirstUncompletedRow() /
           (float)(_n));
  }

  /** Print the current state of the CA. */
  void printCells() {
      clrscr();
      for (int y=_max-1; y>=0; y--) {
          for (int x=0; x<_n; x++) {
              switch( getState(x,y) ) {
                  case EMPTY:
                      cout << " "; break;
                  case RIGHT:
                      cout << "r"; break;
                  case LEFT:
                      cout << "l"; break;
                  case RIGHT_TLB:
                      cout << "R"; break;
                  case LEFT_TRB:
                      cout << "L"; break;
              }
          }
          cout << endl;
      }
  }

  /** Print some statistics. */
  void printStatistics() {
      cout << endl;
      cout << "n = " << _n << " ; max = " << _max
           << " ; no of clocks = " << _clocks << endl;

      int sum = 0;
      for (int y=0; y<_max; y++) {
          for (int x=0; x<_n; x++)
              if (getState(x,y) != EMPTY) sum++;
      }

      float average = sum / (float)_n;

      int noOfCompletedRows = getNoOfCompletedRows();

      int itemsOnFirstUncompletedRow =
          getNoOfNonEmptyCellsOnFirstUncompletedRow();

      int itemsInFinalState = 0;
      for (int y=0; y<_max; y++) {
          for (int x=0; x<_n; x++)
              if (getState(x,y) == RIGHT_TLB) itemsInFinalState++;
      }

      cout << "sum = " << sum << " ; average = "
           << average << endl
           << "no of completed rows = "
           << noOfCompletedRows
           << " ; no of non-empty cells on the first "
           << "uncompleted row = k = "
           << itemsOnFirstUncompletedRow
           << " ; k / n = "
           << (itemsOnFirstUncompletedRow / (float)_n)
           << endl
           << "percentage of cells in final state = "
           << (itemsInFinalState / (float)sum * 100)
           << " % ; "
           << _changed << " cells changed" << endl;
  }

  /** Execute one step. */
  void clock() {
      _clocks++;
      // execute all substeps in the appropriate order
```

```
        subClock1();
        subClock2();
        subClock3();
        subClock4();
        subClock5();
        cleanUp();
    }

private:
    /** Returns the state of the specified cell. */
    int getState(int x, int y) {
        return _cells->getState(x,y);
    }

    /** Sets the state of the specified cell. */
    void setState(int x, int y, int newState) {
        _cells->setState(x,y,newState);
    }

    /** Substep according to Rule 1 (down). */
    void subClock1() {
        Cells *cellsNew = new Cells(_n,_max);
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                cellsNew->setState(x,y,getState(x,y));
                if (getState(x,y) == EMPTY) {
                    if (getState(x,y+1) > EMPTY)
                        cellsNew->setState(x,y,-RIGHT);
                }
                if (getState(x,y) > EMPTY) {
                    if (getState(x,y-1) == EMPTY)
                        cellsNew->setState(x,y,-EMPTY);
                }
            }
        }
        delete _cells;
        _cells = cellsNew;
    }

    /** Substep according to Rule 2 (down-right). */
    void subClock2() {
        Cells *cellsNew = new Cells(_n,_max);
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                cellsNew->setState(x,y,getState(x,y));
                if (getState(x,y) == EMPTY) {
                    if (getState(x-1,y+1) > EMPTY)
                        cellsNew->setState(x,y,-RIGHT);
                }
                if (getState(x,y) > EMPTY) {
                    if (getState(x+1,y-1) == EMPTY)
                        cellsNew->setState(x,y,-EMPTY);
                }
            }
        }
        delete _cells;
        _cells = cellsNew;
    }

    /** Substep according to Rule 3 (down-left). */
    void subClock3() {
        Cells *cellsNew = new Cells(_n,_max);
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                cellsNew->setState(x,y,getState(x,y));
                if (getState(x,y) == EMPTY) {
                    if (getState(x+1,y+1) > EMPTY)
                        cellsNew->setState(x,y,-RIGHT);
                }
                if (getState(x,y) > EMPTY) {
                    if (getState(x-1,y-1) == EMPTY)
                        cellsNew->setState(x,y,-EMPTY);
                }
            }
        }
        delete _cells;
        _cells = cellsNew;
    }

    /** Substep according to Rule 4 (right). */
    void subClock4() {
        Cells *cellsNew = new Cells(_n,_max);
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                int currentState = getState(x,y);
                cellsNew->setState(x,y,currentState);
                if (currentState == EMPTY) {
                    int leftState = getState(x-1,y);
                    if ((leftState == RIGHT) ||
                        (leftState == RIGHT_TLB))
                        cellsNew->setState(x,y,-leftState);
                }
                int rightState = getState(x+1,y);
                if (currentState == RIGHT) {
                    if (rightState >= EMPTY)
                        cellsNew->setState(x,y,-rightState);
                    else if (rightState == OUT_OF_BOUNDS)
                        cellsNew->setState(x,y,-LEFT_TRB);
                }
                if (currentState == RIGHT_TLB) {
                    if (rightState == EMPTY)
                        cellsNew->setState(x,y,-EMPTY);
                }
            }
        }
        delete _cells;
        _cells = cellsNew;
    }

    /** Substep according to Rule 5 (left). */
    void subClock5() {
        Cells *cellsNew = new Cells(_n,_max);
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                int currentState = getState(x,y);
                cellsNew->setState(x,y,currentState);
                if (currentState == EMPTY) {
                    int rightState = getState(x+1,y);
                    if ((rightState == LEFT) ||
                        (rightState == LEFT_TRB))
                        cellsNew->setState(x,y,-rightState);
                }
                int leftState = getState(x-1,y);
                if (currentState == LEFT) {
                    if (leftState >= EMPTY)
                        cellsNew->setState(x,y,-leftState);
                    else if (leftState == OUT_OF_BOUNDS)
                        cellsNew->setState(x,y,-RIGHT_TLB);
                }
                if (currentState == LEFT_TRB) {
                    if (leftState == EMPTY)
                        cellsNew->setState(x,y,-EMPTY);
                    else if ((leftState == OUT_OF_BOUNDS) ||
                             (leftState == RIGHT_TLB))
                        cellsNew->setState(x,y,-RIGHT_TLB);
                    else cellsNew->setState(x,y,-LEFT_TRB);
                }
            }
        }
        delete _cells;
        _cells = cellsNew;
    }
```

```
    /** Reset the markers. */
    void cleanUp() {
        _changed = 0;
        for (int x=0; x<_n; x++) {
            for (int y=0; y<_max; y++) {
                if (getState(x,y) < 0) {
                    _changed++;
                    setState(x,y,abs(getState(x,y)));
                }
            }
        }
    }

    /** Returns the number of completely filled rows. */
    int getNoOfCompletedRows() {
        int noOfCompletedRows = 0;
        for (int y=0; y<_max; y++) {
            bool complete = true;
            for (int x=0; x<_n; x++)
                if (getState(x,y) == EMPTY)
                    complete = false;
            if (complete) noOfCompletedRows++;
            else break;
        }
        return noOfCompletedRows;
    }

    /**
       Returns the number of non-empty cells on the first
       uncompleted row.
    */
    int getNoOfNonEmptyCellsOnFirstUncompletedRow() {
        int noOfCompletedRows = getNoOfCompletedRows();
        int itemsOnFirstUncompletedRow = 0;
        if (noOfCompletedRows < _max) {
            for (int x=0; x<_n; x++)
                if (getState(x,noOfCompletedRows) != EMPTY)
                    itemsOnFirstUncompletedRow++;
        }
        return itemsOnFirstUncompletedRow;
    }


    int _n; // the number of integers
    int _max; // the maximum integer
    int _clocks; // the number of already executed steps

    // the number of cells whose state has changed during
    // the last step
    int _changed;
    float _desiredValue; // the average of all integers

    Cells *_cells; // the cells of this automaton
};


/**
   Runs a randomly generated test case automatically and
   checks the result.
*/
void runAutomaticly() {
    int n = 120; int m = 100;

    int input[n];
    for (int i=0; i<n; i++) {
        input[i] = (int)(rand() /
                         (double)(RAND_MAX+1.0) * m);
    }

    AverageCA* ca = new AverageCA(n, input);

    char ch;
    //ca->printCells();
    //ca->printStatistics();
    //cin.get(ch);
    do {
        ca->clock();
        //clrscr();
        //ca->printStatistics();
    } while ( ! ca->isFinished());

    //ca->printCells();
    //ca->printStatistics();
    cout << "DESIRED VALUE = " << ca->getDesiredValue()
         << " ; ACTUAL VALUE = " << ca->getActualValue()
         << endl;
    if (abs(ca->getDesiredValue() -
            ca->getActualValue()) > 0.00001) {
        cout << "ERROR !!!" << endl;
        exit(-1);
    }
    else {
        cout << "OKAY" << endl;
    }
    //cin.get(ch);

    delete ca;
}


/**
   Runs a randomly generated test case.
   After each step the user has to press RETURN.
*/
void runManually(int n, int m) {
    int input[n];
    for (int i=0; i<n; i++) {
        input[i] = (int)(rand() /
                         (double)(RAND_MAX+1.0) * m);
    }

    AverageCA* ca = new AverageCA(n, input);

    char ch;
    ca->printCells();
    ca->printStatistics();
    cout << "(Press RETURN to go on or enter \''q\'' and "
         << "press RETURN to quit.)" << endl;
    cin.get(ch);
    while(ch != 'q') {
        ca->clock();
        ca->printCells();
        ca->printStatistics();
        cout << "(Press RETURN to go on or enter \''q\'' and "
             << "press RETURN to quit.)" << endl;
        cin.get(ch);
    }
    delete ca;
}

/** The main method. */
int main() {
    // specify here the number of integers n
    // and the maximum value of the integers m
    int n = 30; int m = 11;

    runManually(n,m);


    return 0;
}
```