The Mean Median Map

Miroslav Bonchev Bonchev

May 4, 2011

MSci Project

Queen Mary, University of London

Student #: 075039325

Contact: miro@mbbsoftware.com
ah07278@qmul.ac.uk (until end of June 2011)
http://www.mbbsoftware.com

Abstract: The mean median map is a recursive function with complex behaviour. The paper explains the nature of the function. It also presents a new numeric library for arbitrary large integer, arbitrary precise rational numbers and irrational numbers created for this investigation. The paper introduces the Object Specialization Model for first time, which although created previously was used in the software and hence required explanation. The Object Specialization Model adds to the abstraction of the model of Object Oriented programming enhancing it and making software development safer and faster.

Contents

I Introduction 4

II Background and Conjectures 6

2.1. Definition of the problem 6

2.2. Conjectures 7

III Results 9

IV Software 27

4.1 Object Specialization Model 27

4.2 Proper Numbers Library 36

4.2.1. The Integer Class 36

4.2.2. The Rational Class Template 53

4.2.3. The Quadratic Irrational Template Class 61

4.3. The Mean Meadin Map Application 74

4.4. Matlab Resources 92

4.5. Other Classes and Resources 93

V Conclusion 94

List Of Figures

1. Start Value vs. Halt Value for sequences with starting sets $\{0,\, x,\, 1\}$, $x = 0.5 + \frac{1}{20000}n,\ 0 \leq n \leq 10000,\ n \in \mathbb{N}$ 14
2. Start Value vs. Iteration Count Until Halt for sequences with starting sets $\{0,\, x,\, 1\}$, $x = 0.5 + \frac{1}{20000}n,\ n = 1..10000.$ 15
3. Zoom in on the interval 0.514 - 0.515 with starting sets $\{0,\, x,\, 1\}$, $x = 0.514 + \frac{1}{10000000}n,\ 0 \leq n \leq 10000,\ n \in \mathbb{N}.$ 15
4. Plot of sequences until the halt, for starting set $\{0,\, 0.5(2180),\, 1\}$. 20
5. Halt Kernel forming for sequence with starting set $\{0,\, 0.7,\, 1\}$. 21
6. Data - final iteration - produced by the software for sequence with starting set $\{0,\, 0.7,\, 1\}$. 21
7. Plot of sequences until they halt. All over 20,000 experiments yield similar results. 25
8. Class diagram of the specialize< parameters > template. 29
9. Class diagram of the Integer class from the Proper Numbers Library. 37
10. Class diagram of the RationalTBBase template and Rational specialization type from the PN Library. 53
11. Class diagram of the Functional and Quadratic template classes. 61
12. Class diagram of the Functor abstract class and two derived from it functors. 62
13. Work threads data transmission class diagram. 75
14. Class diagram of the DataSource abstract class and the derived from it Data Sources. 75

List Of Program Listings

1. First revision of the main loop of a function instantiating the mean median map. 10
2. Improved first revision of the mean median map instantiating function to use definitive terminating condition. 14
3. Improved revision of the main loop of the mean median map instantiating function. 19
4. Template class achieving variable specialization using already available C++ facilities. 28
5. Implementation of the Object Specialization Model using templates and inheritance. 34
6. The Integer class from the Proper Numbers Library. 38
7. The RationalTBase template and Rational specialization type from the Proper Numbers Library. 54
8. The Quadratic and Functional templates, and the Functor, SquareRoot and CubeRoot classes. 63
9. The Mean Median Map Application. 76
10. MATLAB script for importing and visualising data from multiple sequences. 92
11. MATLAB script for importing and visualising data of a Mean Median Map sequence. 93

Part I

Introduction

The mean median map is a simple to define function, which demonstrates very complex behaviour and spectacular dynamics. The mean median map starts with a finite sequence, which after number of recursive iterations always seem to settle to some value. To settle to some value means that every new value produced by the map is the same as the one before. The mean median map has been suggested by Schultz and Shiflett Ref. [2] and has been further investigated by number of mathematicians, including Chamberland and Martelli Ref. [1]. These authors have proposed number of conjectures regarding the nature and behaviour of the mean median map. The author of this paper will answer to some of the questions which they raise. For example in Ref. [1] Chamberland and Martelli notice a remarkable coincidence of a double occurrence of the settling value some time before the settling occurs. These 'miraculous' appearances seem to occur randomly. This paper will demonstrate that in fact in order for the sequence to settle there must be such double occurrence and in fact that this is the condition for settling of the sequence. Although relatively rare there are many sequences in which the repeated settling value occurs three or more times before the sequence settles. It will be also demonstrated that these occurrences need not be isolated or unique, and that in fact the settling value is the smallest value of all such repeated values. For the successful outcome of this investigation it was necessary for the author to develop substantial amount of software, which is intricate to, and undividable part from the problem solution and therefore it is presented in the paper as integral part of it.

The positive outcome of this investigation was mostly due to the complementary and mutually beneficial use of mathematics and computer science. To write the software required to help the investigation the author had to first deduce and prove the sequence settling condition. Clearly two or more equal consecutive values produced by the map are necessary but not a sufficient condition to say that the sequence is settled to that value. Hence mathematics produced the important theorem that allowed the software to produce settling values at all, i.e. a terminating condition. On the other hand the powerful and flexible software written for this project allowed the author to produce a large number of converging experiments which allowed important observations and inspired number of theorems. These theorems on the other hand then allowed the software to be improved, thus a later version of the software using them is approximately ten times faster than the previous versions not using them, which in turn allowed even more experiments. However, the more important contribution of computer science to this project was the perspectives that it brought to the table allowing the decomposition of the problem to a lower level of what mathematics usually settles for, and allowing the author to see the intricacy of the map and explain it functioning. In particular these were the definitions of the Sorted Sequence, Centre of the Sorted Sequence, Halt Kernel, Critical Distance and others. In addition to these abstract definitions in order to be able to compute the settling value of the map for an arbitrary rational number the author had to develop an arbitrary precision rational numbers since the floating point numbers are unable to satisfy the conditions for absolute precision. Arbitrarily precise rational numbers however are only possible after first creating arbitrarily large integer numbers. Starting the project with rational numbers defined over 64 bit integers for both the numerator and the denominator it became soon clear that much larger precision is required. This lead to the development of an arbitrarily large integer numbers which were used to replace the 64 bit integers in the definition of the rational numbers. On average rational numbers based on 64 bit numerator and 64 bit denominator overflow after about 100 iterations of a non-setteled map, compare this to the 334,562 iterations required for the map to settle when the starting sequence is (0, 5141309/9999000, 1). The software was later expanded to use quadratic irrational and other irrationals with similar form. The generic numbers developed for this project are now aggregated in a generic numeric library called **Proper Numbers Library.** The Proper Numebrs library offers arbitrarily large integer numbers, arbitrary precise rational numbers and a set of irrational numbers with the form of quadratic irrationals. It is published under the MIT Open Source Software License Agreement, can be downloaded from `http://www.MBBSoftware.com/Software/ProperNumbersLibrary/Default.aspx` and freely used by mathematicians, computer scientists and software engineers.

In addition to the insights about of the median map and although not necessarily related to it, this paper also introduces a new software methodology called **Object Specialization Model.** The author developed the Object Specialization Model previously, but due the tight time frames for this investigation it was used in the developed software in order to achieve (a) faster devleopment with less errors and (b) more coherent code and in particular better Proper Numbers Library. Since the Object Specialization Model is a new concept, published here for first time it is necessary to be explained. The Object Specialization Model makes software development more comprehensive and at the same time faster due reducing the possibility for making errors. This is achieved by adding context to objects (variables) and thus making impossible to miss-assign variables, which is one of the common mistakes during software development. For example, consider the two sequences used in the median map: the sequence of values that are produced by the map and the sequence of the same values but sorted in ascending order. Generally, the two sequences are from the same type, say Rational, so the author could easily make the mistake to add a value to the wrong sequence. This would lead to investigating a different map, with different properties and invalid for the mean median map results until the error is located and corrected. By using the Object Specialization Model however the author adds context to each of the sequences, which makes any such invalid assignment or arithmetic operation intrinsically impossible. A simpler to understand example would be the following one: suppose that there are two Boolean variables representing the values of two unrelated propositions say: “Today is Monday” and “Water flows”. Clearly they have nothing in common and must not be confused or mixed, however since the two variables representing the propositions are of the same Boolean type it is possible to erroneously assign the value of one of them to the other or use them in the same Boolean expression. The Object Specialization Model makes such errors impossible. The Object Specialization Model has additional architectural benefits such as expanding and structuring the namespace and language syntax function space expansion which is otherwise impossible and others. The Object Specialization Model requires no additional programming, presents no performance overhead, but requires somewhat more typing. It is available from `http://www.MBBSoftware.com/Software/ObjectSpecializationModel/Default.aspx` under the MIT Open Source Software License Agreement.

Part II

Background and Conjectures

To understand the nature and behaviour of the Mean Median Map the author developed fair amount of software and applied formal software development techniques. It is important to stress that in this paper the software development is not a serf with void significance of its implementation. Quite to the contrary, formal methods and approaches from computer science and mathematics flow freely in both directions and are equally important. Precisely this approach is what allowed the author to comprehend the nature of the mean-median map functions.

2.1. Definition of the problem

The median of a set $S_n = \{x_1,\, x_2,\, x_3,\, \ldots,\, x_n\}$, where $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$, is defined (traditionally) as

$$median(S_n) := \begin{cases} x_{\frac{n+1}{2}}, & n - odd \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}), & n - even \end{cases} .$$

for simplicity the median will be denoted as

$$M(S_n) := median(S_n)\,,$$

and when there is no ambiguity as

$$M_n := median(S_n)\,.$$

Definition. The recursive sequence X is denoted as

$$X = (x_n), \quad with \quad X_n = (x_1, x_2, ..., x_n),$$

where x_n is defined as the solution of the *mean-median equation*

$$\frac{x_1 + x_2 + x_3 + \ldots + x_n}{n} = M_{n-1}\,, \tag{0.1}$$

and is specified by a list of initial values x_1, x_2, ..., x_{n-1}.

The following definition is required by the necessity of the existence of the object that it describes in any software model attempting to implement X_n as defined above.

Definition. The sequence X_n^{sort} is obtained from X_n by arranging its terms in ascending order such that $\tilde{x}_n \in X_n^{sort}$, where $\tilde{x}_i \leq \tilde{x}_{i+1}$ for all i, $0 < i < n$ using the function

$$\Xi : \mathbb{R} \times X_n \to X_n^{sort}. \tag{0.2}$$

In particular after each iteration and generation of a new $x_n \in X_n$, x_n is also placed in ascending order as $\tilde{x}_k \in X_n^{sort}$, k$\leq n$ by the sorting function. The function Ξ is essentially a permutation on X_n defined by ascending in $\mathbb{R}$ order. The use of tilde instead of superscript to denote the elements of the ordered sequence is in order to reserve the superscript for use allowing distinguishing between the elements of different iterations $x_k^{[n]}$, where $k \leq n$ and $[n]$ denotes the iteration count i.e. the number of elements in X_n and respectively $\tilde{x}_k^{[n]}$ for X_n^{sort}.

The sorted sequence is necessary to compute the median. Normally its existence is understood as implied, overlooked and largely ignored.

Definition. A sequence X_n produced by the rule 0.1 is said to be **halted** if $\exists j \in \mathbb{N}$, $\forall k \in \mathbb{N}$, $k > j$, s.t. $x_j = x_k$, and x_j is called **halt value**.

The sum of all elements of the sequence is:

$$S_n = \sum_{i=1}^{n} x_i \ .$$

The mean value (expectation) of the sequence is given by:

$$E_n = \frac{S_n}{n} = \frac{1}{n} \sum_{i=1}^{n} x_i \ .$$

Using this notation we can now express the definition 0.1 of the sequence X_n in the following equivalent ways:

$$x_{n+1} = (n+1)M_n - \sum_{i=1}^{n} x_i = (n+1)M_n - S_n = (n+1)M_n - nE_n \tag{0.3}$$

2.2. Conjectures

Chamberland and Martelli have formalized the following conjectures.

Conjecture 1. *(Chamberland and Martelli - 2007 (Conjecture 2.1)).* Strong terminating conjecture

For every finite non-empty set $S \subset \mathbb{R}$, there exists an integer k such that the associated infinite sequence satisfies $x_j = x_k$, for all $j > k$. In other words the sequence of the terms settles permanently (halts) to the median after finite number of mean-median iterations.

The author of this paper believes that the Strong Terminating Conjecture is true. Since Chamberland and Martelli have used the software package Maple for their investigation, it is likely that they have experienced problems to observe X_n settling for some particular rational numbers. Maple is an interpreter and runs significantly slower relative to compiled and optimized code such as the one created for this investigation. In addition since the number of iterations before the sequence settles is unknown, when using Maple, one would need to allocate significant amount of system memory in order to ensure that there will be enough space for all elements of the sequence in the data containing structure. Hence sufficient amount of system memory for some sequences may have not been available to them at the time of their research. The author of this paper has observed cases when the allocated system memory for certain sequences has been larger than 5 GB, an amount of memory not commonly available in previous years. Further certain theorems suggested in this paper are required to accelerate the computations. Finally the machine used by the author of this paper is of order of magnitude faster than a common PC presently and it is likely to have been also much faster than the machines available to Chamberland and Martelli. In all these conditions of compiled, and optimized as both algorithms and as compilation code running on a particularly fast computer it took several days for some sequences to halt. Chamberland and Martelli suggested the following weaker conjecture.

Conjecture 2. *(Chamberland and Martelli - 2007 (Conjecture 2.2)).* Weak terminating conjecture

For every finite non-empty set $S \subset \mathbb{R}$, the limit of medians is finite.

Conjecture 3. *(Chamberland and Martelli - 2007 (Conjecture 2.3)).* Continuity conjecture

The function M is continuous.

While it seems that this conjecture is also true the author will not take position on it in this paper.

Part III

Results

We will begin with the following fundamental theorem:

Theorem 4. *(Chamberland and Martelli - 2007 (Theorem 2.1)). The sequence of medians is monotone.*

Proof. Starting with $S_n = \{x_1,\, \ldots,\, x_n\} \subset \mathbb{R}$,we have

$$x_1 + \cdots + x_n + x_{n+1} = (n+1)\, M_n.$$

The next iteration produces

$$x_1 + \cdots + x_n + x_{n+1} + x_{n+2} = (n+2)\, M_{n+1}.$$

Subtracting yields

$$x_{n+2} = (n+1)\left[M_{n+1} - M_n\right] + M_{n+1}. \tag{0.4}$$

If $M_{n+1} \geq M_n$, then $x_{n+2} \geq M_{n+1}$, which in turn forces $M_{n+2} \geq M_{n+1}$ by definition of median. This process may be continued indefinitely producing a sequence of medians which is monotonically non-decreasing. Similarly, if $M_{n+1} \leq M_n$, the sequence of medians is non-increasing.

□

The following definition is a direct consequence from formal methods used in software development.

Definition. The centre C_n of the finite sequence X_n^{Sort} is defined

$$C_n := \{\tilde{x}_k :\; k \in K_n\}, \tag{0.5}$$

where K_n is a sequence of indices defined as follows:

$$K_n := \begin{cases} \frac{n+1}{2}, & if\, n\, is\, odd\, and\, X_n\, is\, not\, halted \\ \frac{n}{2},\;\frac{n+1,}{2} & if\, n\, is\, even\, and\, X_n\, is\, not\, halted\; . \\ (j:\; x_j = M_n), & if\, X_n\, is\, halted \end{cases}$$

In computational form the elements of a non-halted sequence that constitute the centre is

$$K_n := \left\{\frac{1}{2}\left[(n+1) - (n+1)\, mod\, 2\right],\; \frac{1}{2}\left[(n+1) + (n+1)\, mod\, 2\right]\right\}$$

and the median defined using the centre is

$$M_n = \frac{1}{|K_n|}\sum_{k=1}^{|K_n|}\tilde{x}_k = \frac{1}{2}\left[x_{\frac{1}{2}[(n+1)-(n+1)\, mod\, 2]} + x_{\frac{1}{2}[(n+1)+(n+1)\, mod\, 2]}\right].$$

Remark. Informally, the centre of a sequence consists from the elements of the sorted sequence which are used to compute the median. When the sequence is halted then the centre consists from all elements equal to the median.

Fact 5. *(Chamberland and Martelli - 2007 (3. Three Initial Points)). The dynamics of the set* $S = \{a,\ b,\ c\}$*, with* $a \le b \le c$*, are equivalent to those of* $\{0,\ x,\ 1\}$*, where* $x \in \left[\frac{1}{2},\ 1\right]$.

Proof: Ref.[1]

Note. W.L.O.G., this fact will be used from now on and also base the software on it. This implies that from now all sequences are M_n- increasing sequences, however the results apply to all sequences.

The following excerpt of source code is the main loop of the first revision of a function instantiating the mean median map. This code generates the X_n sequence and the auxiliary X_n^{Sort} sequence. Later this code will be optimized to incorporate theorems to be discussed. For this reason it is only the most essential loop that is included here omitting the trivial details. The source code is self-explanatory and will not be commented on, but note the implied definition of the Centre of the sorted sequence through the coefficients identifying it. Also note that the loop terminating condition is speculative, with very low probability of error. The first most important objective is identifying the appropriate loop terminating condition. The object specialization is discussed in the later chapters.

Listing 1. First revision of the main loop of a function instantiating the mean median map.

```
while( !AreTheLastTwoHundredXnTheSame() )
{
   MEDIAN_TYPE mtMed_n_plus_1( 0 );
   MEDIAN_TYPE mtX_n_plus_1 = (listXn_ByValue.GetCount() + 1) * mtMed_n - mtSum_n;


   // Add the new Xn in the time domain sequence.
   listXn_ByTime.AddTail( new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >(
                              specialize< MEDIAN_TYPE, specialize< Xn, Clock > >::Initialize( mtX_n_plus_1 ) ) );


   // The new Xn i.e. X(n+1) is now computed, It is not yet injected in the sequence but, I will
   // add "+1" to the cardinality of the sequences to compute the correct median indexes. This is OK
   // since I insert X(n+1) in the sequence independently of the new median computation.
   const DWORD dwSequenceCount( listXn_ByValue.GetCount() + 1 );

   const DWORD dwMedIndexA( (dwSequenceCount + 1 - ((dwSequenceCount + 1) & 0x01)) / 2 );
   const DWORD dwMedIndexB( (dwSequenceCount + 1 + ((dwSequenceCount + 1) & 0x01)) / 2 );

   MList< specialize< MEDIAN_TYPE, specialize< Xn, Value > > >::MP< specialize< MEDIAN_TYPE, specialize< Xn, Value > > >* pMP(
      listXn_ByValue.GetHeadMP() );

   // Insert dX_np1 in the sequence and find the new Median.
   for( DWORD dwIndex = 1, dwConditions = 0; (0x00000003 != dwConditions) && (NULL != pMP); dwIndex++ )
   {
      if( (0 == (0x00000001 & dwConditions)) && (mtX_n_plus_1 <= *pMP->GetObject()) )
      {
         // Insert median here.
         pMP = listXn_ByValue.AddBehind( pMP->GetPrevious(),
                                         new specialize< MEDIAN_TYPE, specialize< Xn, Value > >(
                                            specialize< MEDIAN_TYPE, specialize< Xn, Value > >::Initialize( mtX_n_plus_1 ) ) );

         dwConditions |= 0x00000001;
      }

      if( dwMedIndexA == dwIndex )
      {
         mtMed_n_plus_1 = *pMP->GetObject();
      }

      if( dwMedIndexB == dwIndex )
      {
         mtMed_n_plus_1 += *pMP->GetObject();

         mtMed_n_plus_1 /= 2;
         dwConditions    |= 0x00000002;
      }

      pMP= pMP->GetNext();
   }
```

```
    if( dwSequenceCount != listXn_ByValue.GetCount() )
    {
        // The mtX_n_plus_1 has not yet been added.
        listXn_ByValue.AddBehind( listXn_ByValue.GetTailMP(),
                                  new specialize< MEDIAN_TYPE, specialize< Xn, Value > >(
                                      specialize< MEDIAN_TYPE, specialize< Xn, Value > >::Initialize( mtX_n_plus_1 ) ) );
    }

    mtSum_n += mtX_n_plus_1;
    mtMed_n  = mtMed_n_plus_1;

    listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >( specialize< MEDIAN_TYPE, specialize< Mn, Clock >
        >::Initialize( mtMed_n ) ) );
}
```

Theorem 6. *Suppose the sequence $S_k = \{x_1,\, x_2, \ldots,\, x_k\}$ is a starting set for the recursive sequence X_n . Then for all $n > k+1$, $x_{n+1} = M_n + n(M_n - M_{n-1})$.*

Proof. By definition we have $x_{n+1} = (n+1)\, M_n - S_n = (n+1)\, M_n - x_n - S_{n-1}$, but $x_n = n\, M_{n-1} - S_{n-1}$, so:

$$x_{n+1} = (n+1)\, M_n - [n\, M_{n-1} - S_{n-1}] - S_{n-1} = (n+1)\, M_n - n\, M_{n-1} + S_{n-1} - S_{n-1},$$

and

$$x_{n+1} = (n+1)\, M_n - n\, M_{n-1}. \tag{0.6}$$

Rearranging the brackets and we get:

$$x_{n+1} = M_n + n\,(M_n - M_{n-1})\,. \tag{0.7}$$

Now, suppose $n+1 = k+1$, then we can define $M_n \equiv M_k$, however we cannot define $M_{n-1} \equiv M_{k-1}$ since $x_k \in X_k$ is independent by assumption. So the for $n+1 = k+1$, $x_{n+1} = M_n + n\,(M_n - M_{n-1})$ is not well defined. Finally, suppose $x_{n+1} \equiv x_{k+1}$ is determined by definition 0.1, then for $n \geq k+2$ we have both M_n and M_{n-1} well defined and so the formula $x_{n+1} = M_n + n(M_n - M_{n-1})$ is well defined for all $n > k+1$. □

Note. This form of the sequence generation law is more useful to understand the problem and will be used more often than the original form.

Theorem 7. *For all $n \in \mathbb{N}$, $E_{n+1} = M_n$.*

Proof. By definition of E_n and S_n we write:

$$E_{n+1} = \frac{S_{n+1}}{n+1} = \frac{x_{n+1} + x_n + S_{n-1}}{n+1}\,.$$

From equation 0.6 we express $x_{n+1} = (n+1)\, M_n - n\, M_{n-1}$ and from equation 0.3 we express $x_n = n\, M_{n-1} - S_{n-1}$. Substituting in the equation above we get:

$$E_{n+1} = \frac{1}{n+1}\left[((n+1)\, M_n - n\, M_{n-1}) + (n\, M_{n-1} - S_{n-1}) + S_{n-1}\right].$$

Expanding the brackets and simplifying yields:

$$E_{n+1} = \frac{1}{n+1}\left[(n+1)\,M_n\right],$$

which after cancellation gives the required result:

$$E_{n+1} = M_n\,.$$

□

Corollary 8. *E_n is monotone.*

Proof. M_n is monotone by theorem 4. $E_{n+1} = M_n$ by theorem 7, so E_n is also monotone. □

Lemma 9. *For all $n \geq 1$, $M_n = E_n \Rightarrow x_{n+1} = M_n = E_n$.*

Proof. From equation 0.3 we write $x_{n+1} = (n+1)\,M_n - S_n = (n+1)\,M_n - n\,E_n = n\,M_n - n\,E_n + M_n = M_n$. □

Theorem 10. *Suppose that for some $n \geq 1$, $M_n = M_{n+1}$. Then, for all $t \in \mathbb{N}$ the following hold:*

a) $M_n = M_{n+t}$

b) $x_{n+2+t} = M_n$

c) $M_{n+1} = E_{n+1}$

Proof. **a)** From the definition of the sequence:

$$x_n = n\,M_{n-1} - S_{n-1}$$

so the n-th and $(n+1)$-st medians are therefore $M_n = \frac{S_n + x_{n+1}}{n+1}$ and $M_{n+1} = \frac{S_{n+1}+x_{n+2}}{n+2}$. Since $M_n = M_{n+1}$,

$$\frac{S_n + x_{n+1}}{n+1} = \frac{S_{n+1} + x_{n+2}}{n+2}$$

so

$$\frac{S_n}{n+1} + \frac{x_{n+1}}{n+1} = \frac{S_{n+1}}{n+2} + \frac{x_{n+2}}{n+2}$$

but

$$S_{n+1} = S_n + x_{n+1}$$

and therefore

$$\frac{S_n}{n+1} + \frac{x_{n+1}}{n+1} = \frac{S_n}{n+2} + \frac{x_{n+1}}{n+2} + \frac{x_{n+2}}{n+2}\,.$$

Multiplying both sides by $(n+1)(n+2)$, we obtain:

$$(n+2)S_n + (n+2)x_{n+1} = (n+1)S_n + (n+1)x_{n+1} + (n+1)x_{n+2}.$$

Solving for x_{n+2}, and simplifying, we obtain:

$$x_{n+2} = \frac{S_n + x_{n+1}}{n+1} = M_n = M_{n+1}. \tag{0.8}$$

Now, informally, since $x_{n+2} = M_n = M_{n+1}$, it follows that x_{n+2} will lie between $\tilde{x}_{n+1}$ and $\tilde{x}_n$, i.e., it will become an element of the centre, which implies that $M_{n+2} = x_{n+2} = M_{n+1} = M_n$.

Formally, since $M_n = M_{n+1}$:

- If n is odd, then $\tilde{x}^{[n]}_{\frac{n+1}{2}} = M_n = M_{n+1} = \frac{1}{2}\left(\tilde{x}^{[n+1]}_{\frac{n+1}{2}} + \tilde{x}^{[n+1]}_{\frac{n+1}{2}+1}\right)$,

but $\tilde{x}^{[n]}_{\frac{n+1}{2}} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}}$ or $\tilde{x}^{[n]}_{\frac{n+1}{2}} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}+1}$, so $\tilde{x}^{[n]}_{\frac{n+1}{2}} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}+1}$, where $\tilde{x}^{[n]}_{\frac{n+1}{2}} \in C_n$ and $\tilde{x}^{[n+1]}_{\frac{n+1}{2}}$, $\tilde{x}^{[n+1]}_{\frac{n+1}{2}+1} \in C_{n+1}$ are the only elements of C_n and C_{n+1} respectively.

- If n is even, then $\frac{1}{2}\left(\tilde{x}^{[n]}_{\frac{n}{2}} + \tilde{x}^{[n]}_{\frac{n}{2}+1}\right) = M_n = M_{n+1} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}}$,

but $\tilde{x}^{[n]}_{\frac{n}{2}} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}}$ or $\tilde{x}^{[n]}_{\frac{n}{2}+1} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}}$, so $\tilde{x}^{[n]}_{\frac{n}{2}} = \tilde{x}^{[n]}_{\frac{n}{2}+1} = \tilde{x}^{[n+1]}_{\frac{n+1}{2}}$, where $\tilde{x}^{[n]}_{\frac{n}{2}}$, $\tilde{x}^{[n]}_{\frac{n+1}{2}} \in C_n$ and $\tilde{x}^{[n+1]}_{\frac{n+1}{2}} \in C_{n+1}$ are the only elements of C_n and C_{n+1} respectively.

Considering equation 0.8 we write $x_{n+2} = M_n = M_{n+1} = \tilde{x}_m = \tilde{x}_{m+1}$ for some $m < n$ in the sorted sequence where $\tilde{x}_m$, $\tilde{x}_{m+1} \in C$ respectively. This however implies that x_{n+2} will be placed just next to $\tilde{x}_m = \tilde{x}_{m+1}$ in X^{sort}_n, hence:

- if m, $m+2$ are odd, then $M_{n+2} = \tilde{x}_{m+2} = M_{n+1} = M_n$;

- if m, $m+2$ are even, then $M_{n+2} = \frac{1}{2}\left(\tilde{x}_{m+1} + \tilde{x}_{m+2}\right) = \tilde{x}_{m+1} = M_{n+1} = M_n$.

So $M_{n+2} = M_{n+1} = M_n$. Let $m = n+1$,then $M_{m+1} = M_m$ then the argument repeats and we get $M_{m+2=n+3} = M_{n+2} = M_{n+1} = M_n$. Then given any $t \in \mathbb{N}$ the argument repeats t times and we have $M_{n+t} = M_n$.

b) From part a) we have that if $M_n = M_{n+1}$ then $M_n = M_{n+t}$ for all $t \in \mathbb{N}$. Suppose then the condition is met and let n be such that for all $t \in \mathbb{N}$ $M_n = M_{n+t}$. Then by theorem 6 we have:

$$x_{n+2+t} = (n+1)\left(M_{n+1+t} - M_{n+t}\right) + M_{n+1+t}\,,$$

but $M_n = M_{n+t}$, so $M_{n+1+t} - M_{n+t} = 0$, so

$$x_{n+2+t} = M_{n+1+t} = M_n \, for \, all \, t \in \mathbb{N}\,.$$

c) Immediately from theorem 7 $E_{n+1} = M_n$, since $M_n = M_{n+1}$ it follows that $E_{n+1} = M_{n+1}$. □

Note. Theorem 10 gives the condition when the sequence X_n halts, which is $M_n = M_{n+1}$ and the halt value which is $h = M_n = M_{n+1}$. $M_n = E_n \Rightarrow x_{n+1} = M_n = E_n$ is by implication another halting condition, obviously it is equivalent to $M_n = M_{n+1}$. The process of halting the sequence demonstrated chronologically is as follow:

1) $M_n = M_{n+1} = M_{n+1+t}$, for all $t \in \mathbb{N}$

2) $M_{n+1} = E_{n+1} = E_{n+1+t}$, for all $t \in \mathbb{N}$

3) $x_{n+2} = M_{n+1} = x_{n+2+t}$, for all $t \in \mathbb{N}$.

Remark. $x_n = x_{n+1}$ is **not** a condition for the sequence to halt. There may be 2 or more equal consecutive x_n values and yet not have the sequence halted.

Thus we now modify the main loop of the function instantiating the mean median map to use theorem 10 a) in the following way:

Listing 2. Improved first revision of the mean median map instantiating function to use definitive terminating condition.

```
while( mtOldMedian != mtMed_n )
{
   // Store the median as Old-Median before computing the new one.
   mtOldMedian = specialize< MEDIAN_TYPE, specialize< Mn, Clock > >::Initialize( mtMed_n );

   /* as before */
}
```

Corollary 11. *Any sequence with starting set containing one or two elements is trivial and halts immediately.*

Proof. Let $\{x_1\}$ be the starting set for some sequence X_n, then $M_1 = x_1$, so $X_2 = (x_1,\, x_2 = 2x_1 - x_1 = x_1)$, so $M_2 = x_1$, so $M_1 = M_2$ and by theorem 10 a) the sequence is halted. Suppose $\{x_1,\, x_2\}$, $x_1 \le x_2$ is the starting set for some sequence X_n, then $M_2 = \frac{1}{2}(x_1 + x_2)$, so $X_3 = \left(x_1,\, x_2,\, x_3 = \frac{3}{2}(x_1 + x_2) - (x_1 + x_2) = \frac{1}{2}(x_1 + x_2) = M_2\right)$. Now, since $x_1 \le x_2$ it follows that $x_3 = M_2 \le x_2$, so $X_3^{Sort} = (x_1,\, x_3,\, x_2)$, so $M_2 = M_3$ so by theorem 10 a) the sequence is halted. □

Remark. Sequences produced from starting sets with three and more elements are highly nontrivial. Consider the highly nontrivial relationship between the Start Value and the Halt Value, and between the Start Value and the Number of Iterations required for the sequense to halt for sequences produced from three point starting set as follow $\{0,\, x,\, 1\}$, $x = 0.5 + \frac{1}{20000}n$, $0 \le n \le 10000$, $n \in \mathbb{N}$ shown on the figures below. Chamberland and Martelli demonstrate certain dependences on a similar graph that they have produced. The red rectangle in figure 1 zooms in on the first bleep in the linear part of the graph visually.

Figure 1. Start Value vs. Halt Value for sequences with starting sets $\{0,\, x,\, 1\}$, $x = 0.5 + \frac{1}{20000}n$, $0 \le n \le 10000$, $n \in \mathbb{N}$.

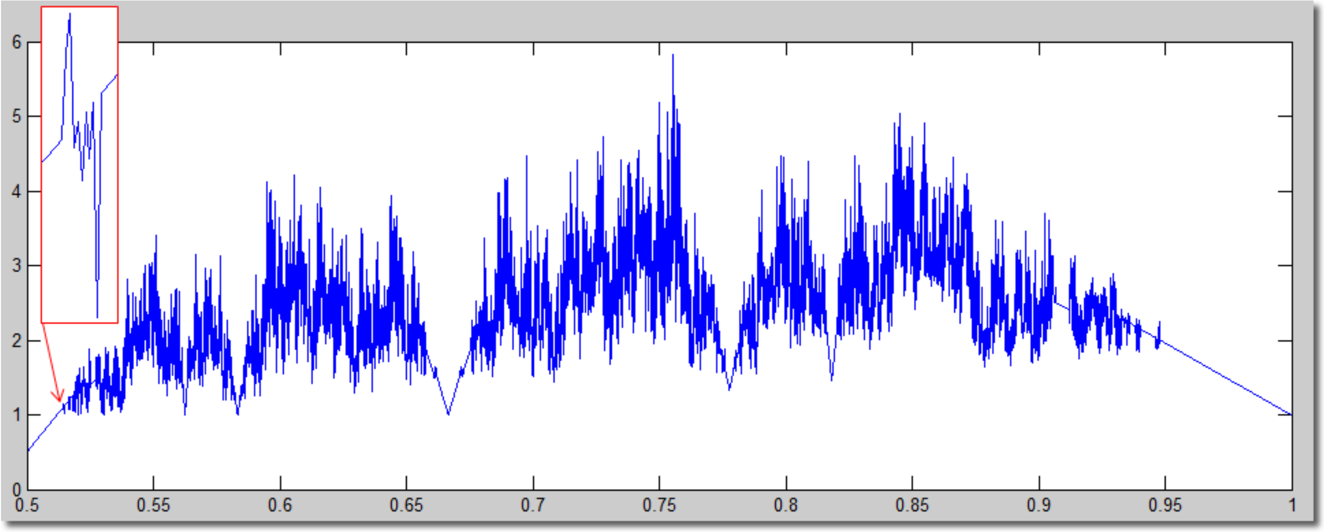
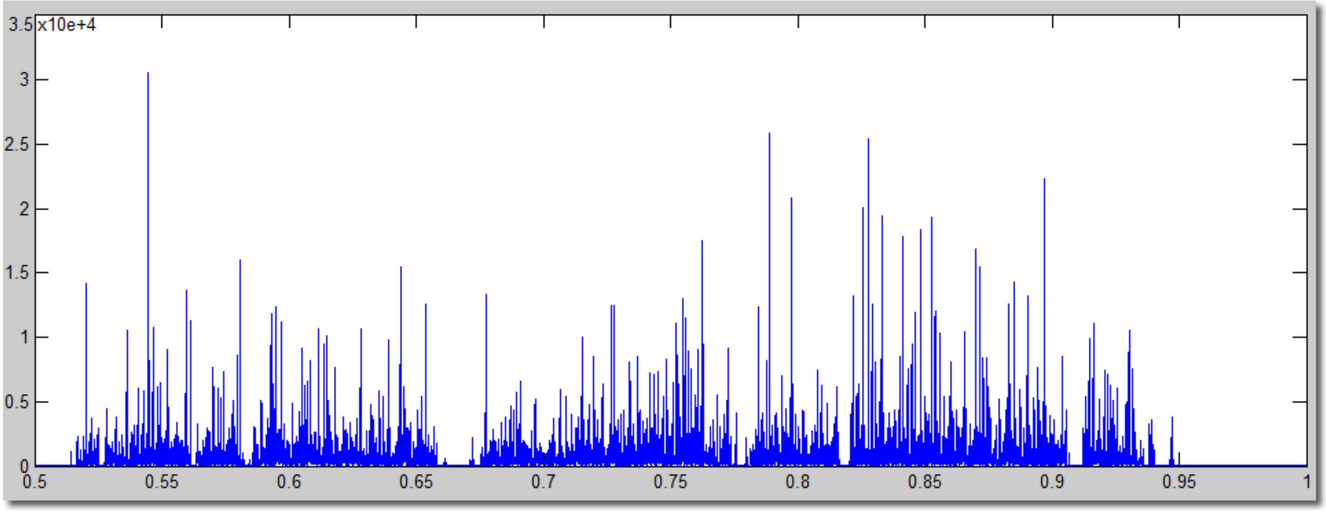
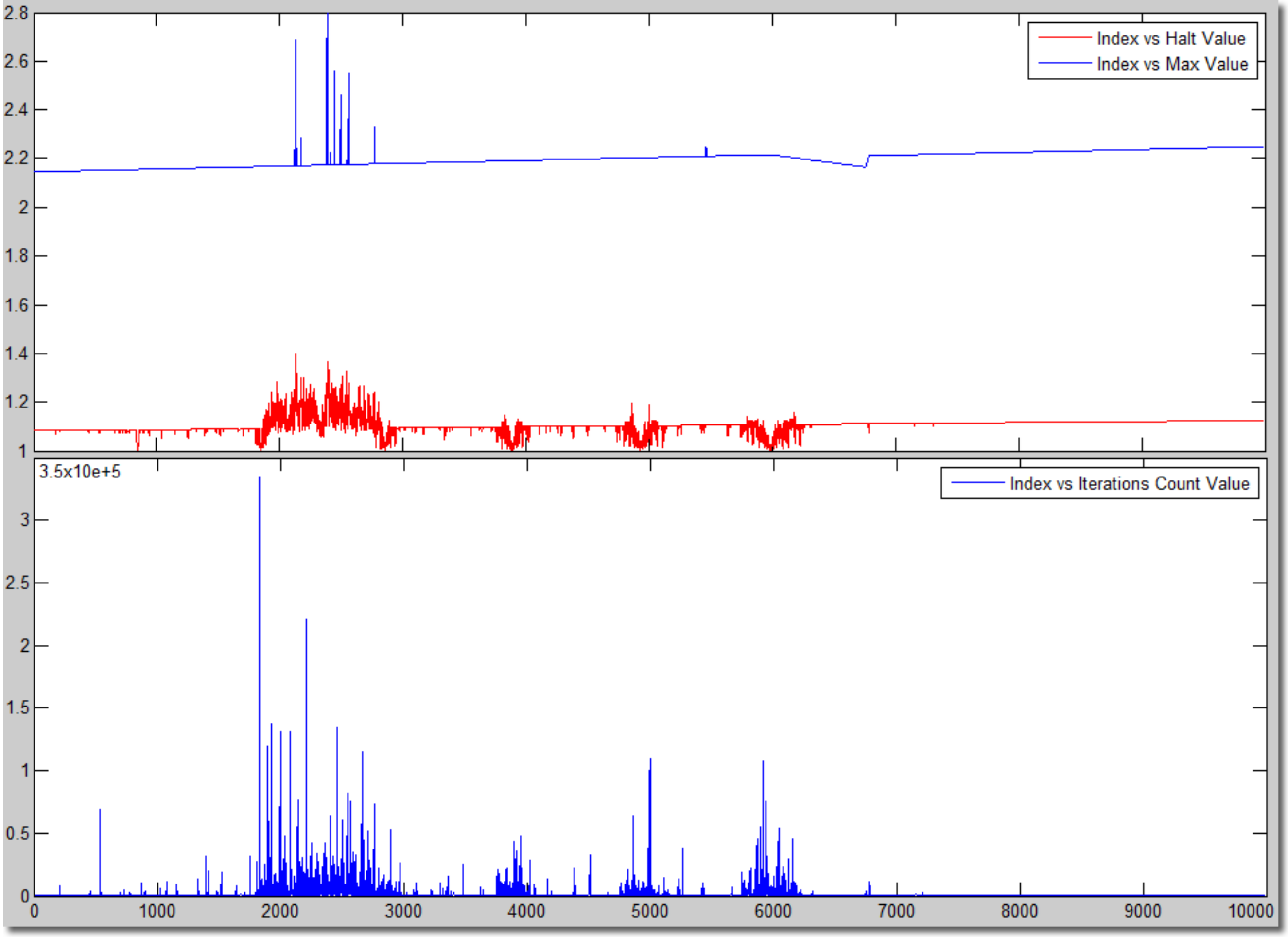


Figure 2. Start Value vs. Iteration Count Until Halt for sequences with starting sets $\{0,\ x,\ 1\}$, $x = 0.5 + \frac{1}{20000}n$, $0 \leq n \leq 10000$, $n \in \mathbb{N}$.



We now zoom in numerically 1000 times on the interval 0.514-0.515 with otherwise the same setting.

Figure 3. Zoom in on the interval 0.514 - 0.515 with starting sets $\{0,\ x,\ 1\}$, $x = 0.514 + \frac{1}{10000000}n$, $0 \leq n \leq 10000$, $n \in \mathbb{N}$.



Observe in figure 2, that the largest number of iterations required for all investigated sequences starting in the interval of the first bleep (0.514-0.515) is approximately 100. In figure 3 however the largest number of iterations for a sequence starting in the same interval is over 300000, in fact 334562 iterations. The more we zoom in, the more complicated graphs are the results. Similar results are produced when experimenting with other numbers, including with sequences converging to an irrational numbers. Number of experiments were produced with sequences converging to the golden ratio -1, i.e. with starting sets $\{0,\ x,\ 1\}$, $x=\frac{F_n}{F_{n+1}}$, where F_n is the n-th Fibonacci number. Clearly the information stored in the graphs is incomplete and not useful for interpretation and solving the problem. We continue with further results.

Corollary 12. *Let $n\in\mathbb{N}$, $M_n=M_{n+1}\Leftrightarrow x_{n+2}=M_{n+1}$.*

Proof. " $\Rightarrow$ " is part b) of theorem 10. " $\Leftarrow$ " Suppose that for some $n\in\mathbb{N}$, $x_{n+2}=M_{n+1}$. From theorem 6we write $M_{n+1}=x_{n+2}=M_{n+1}+(n+1)\left(M_{n+1}-M_n\right)$, so $M_{n+1}-M_n=0$, so $M_{n+1}=M_n$. □

Remark. This corollary may seem trivial, however it is required to safely optimize the software and remove an auxiliary object holding M_{n-1} (or M_n) depending on how the software is constructed. By theorem 10 the sequence halting condition is $M_n=M_{n+1}$, which implies that the software needs to keep track of one M_n and terminate the generation of new x_{n+1} when the current M_n is equal to the previous one. However the above corollary justifies terminating the generation of new x_{n+1} when the newly generated x_{n+1} is equal to the current M_n. This is important for speed wise optimisation of the software. In particular, if one is using processor/coprocessor native arithmetic registers data holders, then the memory spent for one additional M_n is negligible, as well as the processor time for assignment and comparison. The later in a modern processor would be a few processor clocks, say 7 clocks on average, propagated on average say 50,000 elements sequence is still not essential. However since the software is using arbitrarily precise rational and algebraic irrational numbers, constructed over an arbitrarily large integer numbers an assignment and comparison operations may take much larger amount of processor time, and hence this optimisation is important, though it is not essential.

Lemma 13. *Let $k,\ n\in\mathbb{N}$,$k\leq n$, then for all $n\geq 1$, $M_{2n+1}=x_n=\tilde{x}_k$.*

Proof. $2n+1$ is odd for all $n\in\mathbb{N}$, so by definition **0.1** $M_{2n+1}=x_n$, but $\tilde{x}_k$=x_n, for some $k\leq n$. □

Lemma 14. *For all $n\in\mathbb{N}$, $x_{n+1}\geq M_n$.*

Proof. From definition 6 $x_{n+1}=M_n+n(M_n-M_{n-1})$, but M_n is monotone by theorem 4, so $M_n\geq M_{n-1}$, so $n(M_n-M_{n-1})\geq 0$, so $x_{n+1}\geq M_n$. □

Corollary 15. *For every two newly added to a non-halted sequence elements x_{n+1} and x_{n+2} the centre of the sorted sequence travels to the right (towards the greater values elements in X_n^{Sort}) with one position retaining its structure.*

Proof. From lemma 14 $x_{n+1}\geq M_n$ and using the definition of the centre $C_n:=\{\tilde{x}_k\ :\ k\in K_n\}$, where $K_n:=\left\{\frac{1}{2}\left[(n+1)-(n+1)\,mod\,2\right],\ \frac{1}{2}\left[(n+1)+(n+1)\,mod\,2\right]\right\}$. Suppose n is odd, then:

- For K_n, (n is odd) we get:

$K_n=\left\{\frac{1}{2}\left[n+1\right],\ \frac{1}{2}\left[n+1\right]\right\}$

$K_n = \left\{ \frac{n+1}{2} \right\}$.

- For K_{n+1}, ($n+1$ is even) we get:

$K_{n+1} = \left\{ \frac{1}{2} \left[((n+1)+1) - ((n+1)+1) \, mod \, 2 \right], \; \frac{1}{2} \left[((n+1)+1) + ((n+1)+1) \, mod \, 2 \right] \right\}$

$K_{n+1} = \left\{ \frac{1}{2} \left[(n+2) - (n+2) \, mod \, 2 \right], \; \frac{1}{2} \left[(n+2) + (n+2) \, mod \, 2 \right] \right\}$

$K_{n+1} = \left\{ \frac{n+1}{2}, \; \frac{n+3}{2} \right\}$, so the centre position stays the same, but its size increased by one.

- For K_{n+2}, ($n+2$ is odd) we get:

$K_{n+2} = \left\{ \frac{1}{2} \left[((n+2)+1) - ((n+2)+1) \, mod \, 2 \right], \; \frac{1}{2} \left[((n+2)+1) + ((n+2)+1) \, mod \, 2 \right] \right\}$

$K_{n+2} = \left\{ \frac{1}{2} \left[n+3 \right], \; \frac{1}{2} \left[n+3 \right] \right\}$

$K_{n+2} = \left\{ \frac{n+3}{2} \right\}$, moving the position of the centre with one position to the right towards the larger value elements in the ordered by size sequence, and restoring its original structure containing one element.

- For K_{n+3}, ($n+3$ is even) we get:

$K_{n+3} = \left\{ \frac{1}{2} \left[((n+3)+1) - ((n+3)+1) \, mod \, 2 \right], \; \frac{1}{2} \left[((n+3)+1) + ((n+3)+1) \, mod \, 2 \right] \right\}$

$K_{n+3} = \left\{ \frac{1}{2} \left[(n+4) - (n+4) \, mod \, 2 \right], \; \frac{1}{2} \left[(n+4) + (n+4) \, mod \, 2 \right] \right\}$

$K_{n+3} = \left\{ \frac{n+3}{2}, \; \frac{n+5}{2} \right\}$, so the centre position remained the same, but its size increased by one.

Continuing for higher n the pattern repeats itself. So for a non-halted sequence, every two newly added to the sequence elements increase the centre position by one retaining its structure. □

Remark. The statement of this corollary is the same as to as to say that for a non-halted sequence the speed of movement of the centre towards the larger values elements is equal to half of the speed of adding of new elements to the sequence, or that the speed of adding of new elements to the sequence is twice as fast as the speed of the centre.

Lemma 16. *For all $n \in \mathbb{N}$, $x_n \geq M_n$.*

Proof. From lemma 14 $x_{n+1} \geq M_n$, for all $n \in \mathbb{N}$, so x_{n+1} will be positioned to the right (in the direction of increasing values) in the ordered sequence in respect to the centre.

There are the following cases of positioning of x_{n+1} in the ordered sequence. x_{n+1} becomes part of the centre, x_{n+1} is positioned immediately to the right of the centre and finally x_{n+1} is positioned to the right of the centre with some elements of the sequence between the centre and x_{n+1}.

1. x_{n+1} becomes part of the centre.

- if $|X_{n+1}^{Sort}| = 2k+1$, so $|X_n^{Sort}| = 2k$, so $Centre_n = \{a,\, b\}$, so $X_{n+1}^{Sort} = \{...,\, a,\, x_{n+1},\, b,\, ...\}$, so $Centre_{n+1} = \{x_{n+1}\}$, so $M_{n+1} = x_{n+1}$ and in general for this case $x_n = M_n$;

- if $|X_{n+1}^{Sort}| = 2k$, .so $|X_n^{Sort}| = 2k+1$, so $Centre_n = \{a\}$, $X_{n+1}^{Sort} = \{...,\, a,\, x_{n+1},\, ...\}$, so $Centre_{n+1} = \{a,\, x_{n+1}\}$, so $M_{n+1} = \frac{1}{2}\left(a + x_{n+1}\right)$ so $M_{n+1} \leq x_{n+1}$and in general for this case $x_n \geq M_n$;

2. $X_{n+1}^{Sort} = \{...,\, Centre,\, x_{n+1},\, ...\}$, and

- if $|X_{n+1}^{Sort}| = 2k+1$, so $|X_n^{Sort}| = 2k$, so $Centre_n = \{a,\, b\}$ then $M_{n+1} = b$, but $b < x_{n+1}$, so $M_{n+1} < x_{n+1}$ and in general $M_n < x_n$;

- if $|X_{n+1}^{Sort}| = 2k$, so $|X_n^{Sort}| = 2k+1$, so $Centre_n = \{a\}$ then $M_{n+1} = \frac{1}{2}\left(a + x_{n+1}\right)$, but $a < x_{n+1}$, so $M_{n+1} < x_{n+1}$ and in general $M_n < x_n$;

3. $X_{n+1}^{Sort} = \{...,\, Centre,\, ...,\, x_{n+1},\, ...\}$, and for simplicity w.l.o.g. $X_{n+1}^{Sort} = \{...,\, Centre,\, c,\, x_{n+1},\, ...\}$, then

- if $c = x_{n+1}$ we are in the case 1 and we are done;

- if $c \neq x_{n+1}$, then for some $a_i \in Centre$, $i = 1$ or $i = 1,\, 2$ we have $a_i \leq c < x_{n+1}$, then

$$M_{n+1} = \begin{cases} a_1, & |X_{n+1}^{Sort}| = 2k+1, \quad so\ M_{n+1} = a_1 < c < x_{n+1} \\ \frac{1}{2}\left(a_1 + a_2\right), & |X_{n+1}^{Sort}| = 2k, \quad\quad\ \ so\ M_{n+1} = \frac{1}{2}\left(a_1 + a_2\right) < c < x_{n+1}. \end{cases}$$

So $M_{n+1} < x_{n+1}$ and in general $M_n < x_n$. Combining this with (1) and (2) above we have the desired result that for all $n \in \mathbb{N}$, $x_n \geq M_n$. □

Remark. The last two results are very important. They are used to accelerate the software on average about ten times. Since $x_{n+1} = (n+1)M_n - S_n$ is not monotone function, without them the software have to scan the whole of the sorted sequence in order to find the correct location for every newly generated x_{n+1}. Using these results however the software can search only from the last median upwards, thus reducing the comparison operations to at least half. Later it will be shown that in fact as a sequence grow in size the relative benefit from these results also grows since the newly generated x_{n+1} are becoming closer and closer to M_n. Further optimization of the software due to these results is that because $x_n \geq M_n$, x_n will be placed always on the right (on the side of the increasing by value members of the sequence), but this means that a pointer to the first element of the centre of X_n^{Sort} will remain valid after adding the new element to the sequence. Therefore it is no longer necessary to calculate the indexes of the elements of the centre X_n^{Sort}, and iterate through the list representing the sequence in order to retrieve the centre elements and use them to calculate the new median. Instead, only keeping a pointer to the first element of the centre of X_n^{Sort} and increasing it with one for every two new elements added to the sequence is sufficient to allow us to compute the median in an optimal manner. Without this result the software computed 2157 sequences starting at 514/1000 (0.514) and reaching 1713881/3333000 (0.5142157215721572) for approximately 80 hours. Modifying the software to use the above results computed the same sample of data for approximately 8 hours, thus achieving approximately 10 fold performance increase.

Listing 3. Improved revision of the main loop of the mean median map instantiating function.

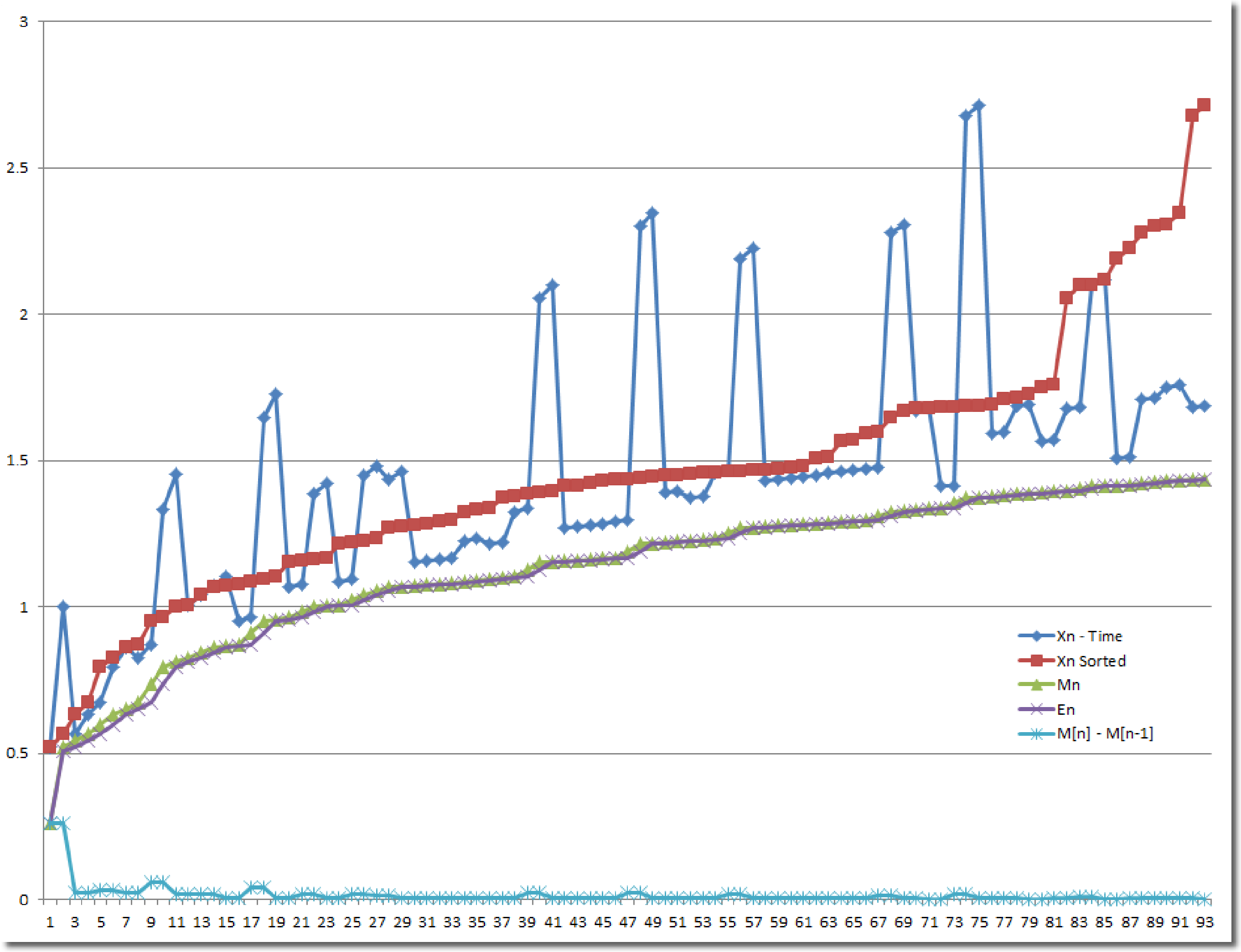
```
for( MEDIAN_TYPE mtX_n_plus_1 = (listXn_ByValue.GetCount() + 1) * mtMed_n - mtSum_n;
     mtX_n_plus_1 != mtMed_n;
     mtX_n_plus_1 = (listXn_ByValue.GetCount() + 1) * mtMed_n - mtSum_n )
{
   // Lemma 12. X[n+1] >= M[n].
   MASSERT( mtX_n_plus_1 >= mtMed_n );


   mtSum_n += mtX_n_plus_1;


   // Add the new Xn in the clock sorted sequence.
   listXn_ByTime.AddTail( new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >(
                              specialize< MEDIAN_TYPE, specialize< Xn, Clock > >::Initialize( mtX_n_plus_1 ) ) );


   // Insert dX_np1 in the sequence and find the new Median.
   for( MList< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > >::MP<
           specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > >* pMP( pMedian ); NULL != pMP; pMP= pMP->GetNext()
              )
   {
      if( mtX_n_plus_1 < pMP->GetObject()->GetLabel() )
      {
         // Insert median here. It is important that the inequality is strict! <= would insert the
         // element in the correct place in respect to the values, but that will make the pointer to
         // the median to point to the wrong place as the sequence will be shifted with one element.
         pMP = listXn_ByValue.AddBehind( pMP->GetPrevious(),
                                         new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >(
                                             specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >::Initialize(
                                                         Pair< MEDIAN_TYPE, DWORD >( mtX_n_plus_1, listXn_ByTime.GetCount() ) ) )
                                                              );

            break;
      }
   }

   if( listXn_ByTime.GetCount() != listXn_ByValue.GetCount() )
   {
      // The mtX_n_plus_1 has not yet been added.
      listXn_ByValue.AddTail( new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >(
                                  specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >::Initialize(
                                              Pair< MEDIAN_TYPE, DWORD >( mtX_n_plus_1, listXn_ByTime.GetCount() ) ) ) );
   }

   if( 0 == (listXn_ByValue.GetCount() & 1) )
   {
      mtMed_n = (pMedian->GetObject()->GetLabelRef() + pMedian->GetNext()->GetObject()->GetLabelRef())/2;
      pMedian = pMedian->GetNext();
   }
   else
   {
      mtMed_n = pMedian->GetObject()->GetLabelRef();
   }

   listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >(
                            specialize< MEDIAN_TYPE, specialize< Mn, Clock > >::Initialize( mtMed_n ) ) );
}
```

Corollary 17. *The sequences X_n and X_n^{Sort} are bounded below by $inf\,(M_n)$, and in particular by M_1.*

Proof. The first part follows immediately from lemma 16. The second part follows from the first part and from the fact that (M_n) is increasing. □

We now display a helpful graph visualizing the above results for X_n, X_n^{Sort}, M_n, E_n and $M_n - M_{n-1}$ in respect to the iteration count until the X_n halts. The starting set for this experiment is $\{0,\ 0.5(2180),\ 1\}$, however the dynamics of this sequence is common for nontrivial sequences. The graph shows the sequences until $M_n \neq M_{n-1}$, which is the terminating condition for the sequence generation as per theorem 10 a) and program listing 3.

Figure 4. Plot of sequences until the halt, for starting set $\{0,\ 0.5(2180),\ 1\}$.



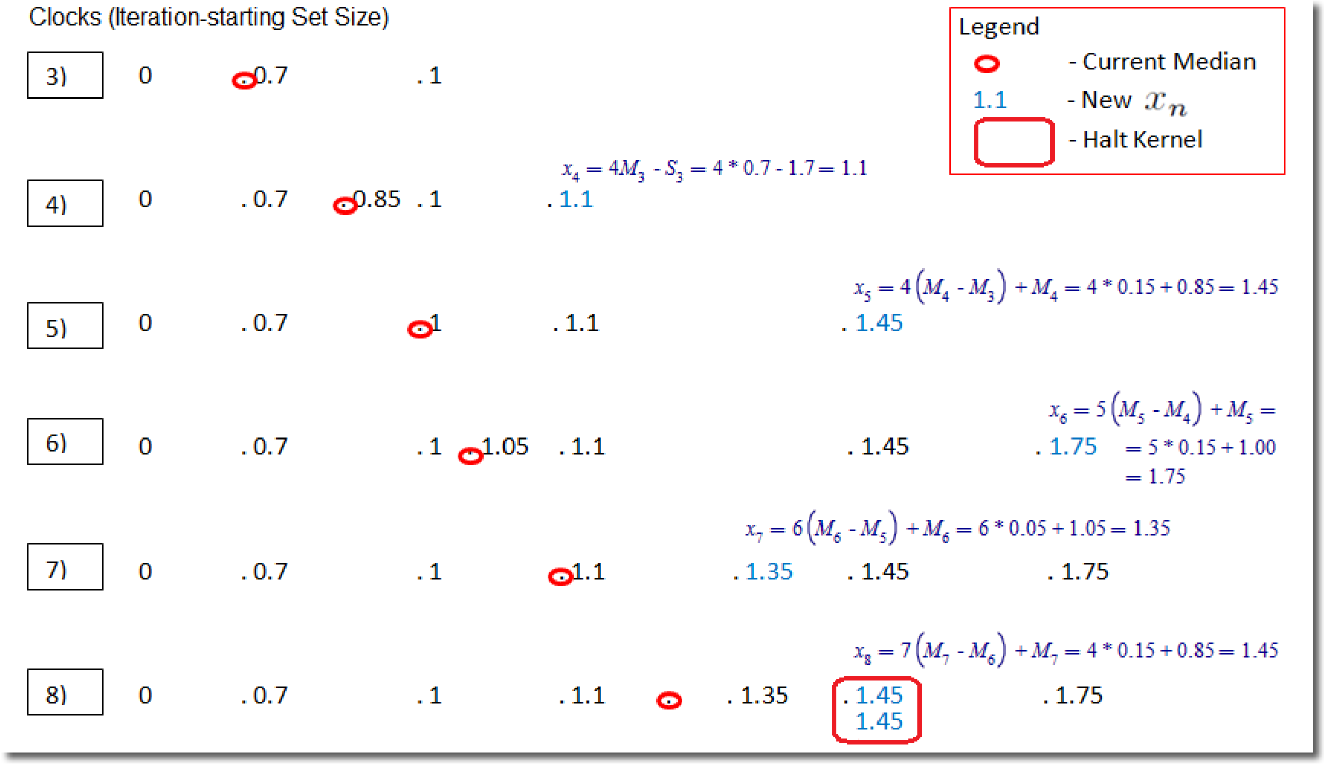
We acknowledge the increasing (monotone) X_n^{Sort}, M_n and E_n sequences and the non-monotone X_n and $M_n - M_{n-1}$, as well as $x_n \geq M_n$ for all $n \in \mathbb{N}$ and other observable results.

Definition. Any subsequence $K := \left(\tilde{x}_i,\ \tilde{x}_{i+t}\ :\ \frac{1}{2}\left[(n+1) + (n+1)\,mod\,2\right] < i,\ 0 < t \in \mathbb{N},\ \tilde{x}_i = \tilde{x}_{i+t}\right) \subset X_n^{Sort}$ is called **Halt Kernel**.

Remark. A halt kernel is a subsequence of X_n^{Sort} such that it has more than one elements which are equal and these elements are bigger than the elements constituting the centre, i.e. they are to the right of the centre (in the direction of increasing values). Since X_n is not monotone suppose that for some $i,\ j < n,\ i \neq j,\ x_i = x_j \neq M_n$, then for some $k \leq n$ we have $x_i = \tilde{x}_k = \tilde{x}_{k+1} = x_j$ and thus $\tilde{x}_k = \tilde{x}_{k+1} \in X_n^{Sort}$ form a halt kernel in it.

The following example illustrates the operation of the map. Suppose we begin with starting set $\{0,\ 0.7,\ 1\}$. The first new element is calculated using definition 0.1 is $x_4 = 1.1$. The next x_n are calculated using theorem 6. In reality the software uses only definition 0.1, however it is easier understanding the mean median map in terms of theorem 6. For this reason on the figure starting from x_5 onwards new x_n are computed using theorem 6. The value produced for x_5 is 1.45. By corollary 15 the centre of the sequence travels with half of the speed of adding

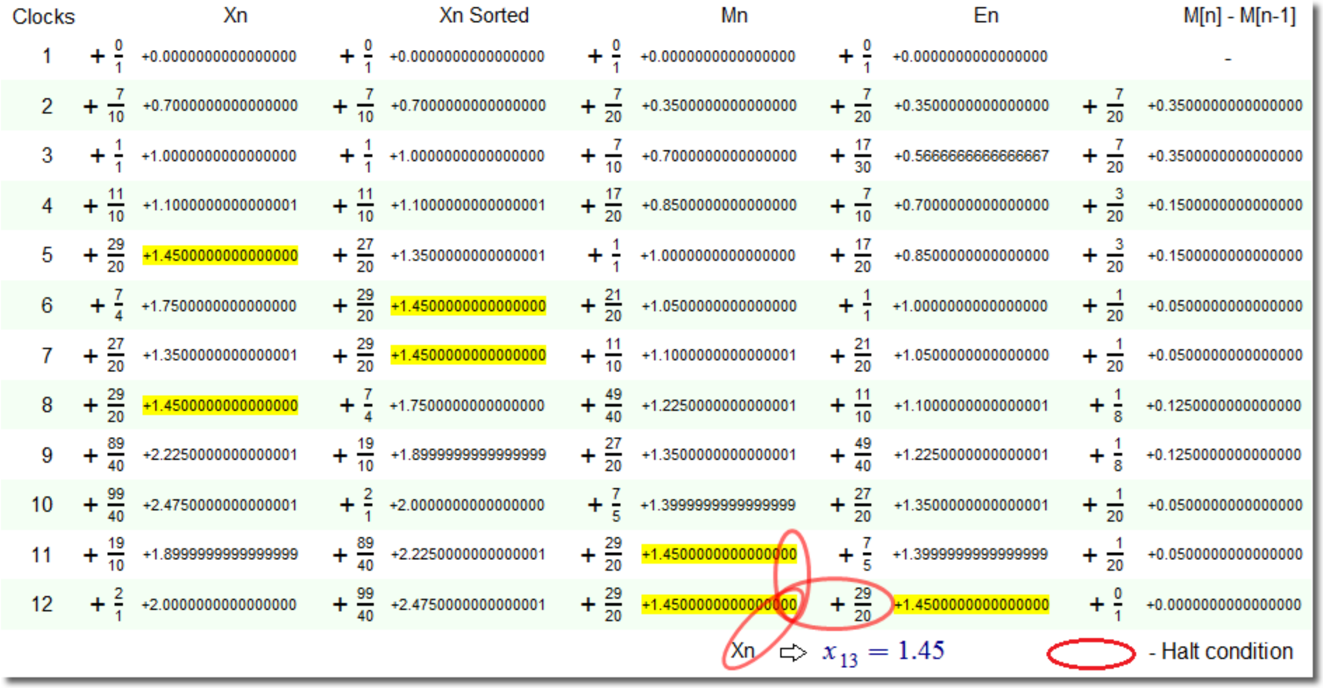
Figure 5. Halt Kernel forming for sequence with starting set $\{0,\ 0.7,\ 1\}$.



Clocks (Iteration-starting Set Size)
3) 0 0.7 . 1
4) 0 . 0.7 0.85 . 1 . 1.1
$x_4 = 4M_3 - S_3 = 4 * 0.7 - 1.7 = 1.1$
5) 0 . 0.7 1 . 1.1 . 1.45
$x_5 = 4\left(M_4 - M_3\right) + M_4 = 4 * 0.15 + 0.85 = 1.45$
6) 0 . 0.7 . 1 1.05 . 1.1 . 1.45 . 1.75
$x_6 = 5\left(M_5 - M_4\right) + M_5 =$
$= 5 * 0.15 + 1.00$
$= 1.75$
7) 0 . 0.7 . 1 1.1 . 1.35 . 1.45 . 1.75
$x_7 = 6\left(M_6 - M_5\right) + M_6 = 6 * 0.05 + 1.05 = 1.35$
8) 0 . 0.7 . 1 . 1.1 . 1.35 . 1.45 1.45 . 1.75
$x_8 = 7\left(M_7 - M_6\right) + M_7 = 4 * 0.15 + 0.85 = 1.45$
Legend
- Current Median
1.1 - New x_n
- Halt Kernel

new elements to X_n. The next two values added to the sequence are $x_6 = 1.75$ and $x_7 = 1.35$. The following $x_8 = 1.45$ however is equal to x_5, so as x_8 is added to the sequence a new Halt Kernel is created. This halt kernel

Figure 6. Data - final iteration - produced by the software for sequence with starting set $\{0,\ 0.7,\ 1\}$.



Clocks	Xn	Xn Sorted	Mn	En	M[n] - M[n-1]
1	$+\frac{0}{1}$ +0.0000000000000000	$+\frac{0}{1}$ +0.0000000000000000	$+\frac{0}{1}$ +0.0000000000000000	$+\frac{0}{1}$ +0.0000000000000000	-
2	$+\frac{7}{10}$ +0.7000000000000000	$+\frac{7}{10}$ +0.7000000000000000	$+\frac{7}{20}$ +0.3500000000000000	$+\frac{7}{20}$ +0.3500000000000000	$+\frac{7}{20}$ +0.3500000000000000
3	$+\frac{1}{1}$ +1.0000000000000000	$+\frac{1}{1}$ +1.0000000000000000	$+\frac{7}{10}$ +0.7000000000000000	$+\frac{17}{30}$ +0.5666666666666667	$+\frac{7}{20}$ +0.3500000000000000
4	$+\frac{11}{10}$ +1.1000000000000001	$+\frac{11}{10}$ +1.1000000000000001	$+\frac{17}{20}$ +0.8500000000000000	$+\frac{7}{10}$ +0.7000000000000000	$+\frac{3}{20}$ +0.1500000000000000
5	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{27}{20}$ +1.3500000000000001	$+\frac{1}{1}$ +1.0000000000000000	$+\frac{17}{20}$ +0.8500000000000000	$+\frac{3}{20}$ +0.1500000000000000
6	$+\frac{7}{4}$ +1.7500000000000000	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{21}{20}$ +1.0500000000000000	$+\frac{1}{1}$ +1.0000000000000000	$+\frac{1}{20}$ +0.0500000000000000
7	$+\frac{27}{20}$ +1.3500000000000001	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{11}{10}$ +1.1000000000000001	$+\frac{21}{20}$ +1.0500000000000000	$+\frac{1}{20}$ +0.0500000000000000
8	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{7}{4}$ +1.7500000000000000	$+\frac{49}{40}$ +1.2250000000000001	$+\frac{11}{10}$ +1.1000000000000001	$+\frac{1}{8}$ +0.1250000000000000
9	$+\frac{89}{40}$ +2.2250000000000001	$+\frac{19}{10}$ +1.8999999999999999	$+\frac{27}{20}$ +1.3500000000000001	$+\frac{49}{40}$ +1.2250000000000001	$+\frac{1}{8}$ +0.1250000000000000
10	$+\frac{99}{40}$ +2.4750000000000001	$+\frac{2}{1}$ +2.0000000000000000	$+\frac{7}{5}$ +1.3999999999999999	$+\frac{27}{20}$ +1.3500000000000001	$+\frac{1}{20}$ +0.0500000000000000
11	$+\frac{19}{10}$ +1.8999999999999999	$+\frac{89}{40}$ +2.2250000000000001	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{7}{5}$ +1.3999999999999999	$+\frac{1}{20}$ +0.0500000000000000
12	$+\frac{2}{1}$ +2.0000000000000000	$+\frac{99}{40}$ +2.4750000000000001	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{29}{20}$ +1.4500000000000000	$+\frac{0}{1}$ +0.0000000000000000

Xn ⇨ $x_{13} = 1.45$ - Halt condition

is the first and only one for this sequence. Since the centre is on the left traveling to the right it approaches the halt centre as more elements are added to the sequence. Once the centre reaches and absorbs the halt centre the sequence halts as there are two equal medians one after another which by theorem 10 a) is the halt condition. The software is able to produce data sheets similar to the one on figure 6 for any iteration allowing the observer to see the Halt Kernel formation and the Centre traveling towards it and absorbing it. For some sequences the map may generate additional halt kernels with smaller or larger value before it reaches any previously generated halt kernel. In the previous example suppose that the value $\mathrm{x}_9 = 1.35$ instead of 2.225, then we would have a new smaller halt kernel generated just before to be absorbed by the centre and halt the sequence. For this map however the next four values are $x_9 = 2.225$, $x_{10} = 2.475$, $x_{11} = 1.8(9)$, $x_{12} = 2.0$ pushing the centre towards the halt kernel $K = \{1.45,\, 1.45\}$. Finally $x_{13} = 1.45 = M_{12}$ showing that the sequence was already halted by corollary 12 and theorem 10.

Definition 18. Critical Distance $\Delta(x_{p-1},\, x_p,\, x_{p+1})$ between three points x_{p-1} and x_p determining M_{n-1} and M_n and x_{p+1} thus $x_{n+1} = M_n + n\,(M_n - M_{n-1})$ is the distance determining if $x_{n+1} \leq x_{p+1}$ or $x_{n+1} > x_{p+1}$.

Theorem 19. *Let X_n be a recursive sequence generated by the mean median map, then $\exists\, m \in \mathbb{N}$ s.t. $\forall\, n \in \mathbb{N}$, $x_n \leq x_m$, we define $x_{maximal} := x_m$.*

Proof. Let X_n be a recursive sequence generated by the mean median map from starting set $\{x_1,\, x_2, \ldots,\, x_k\}$ and suppose $n > k+1$, then recall $x_{n+1} = M_n + n\,(M_n - M_{n-1})$ is true by theorem 6. Define $Base_n := M_n$ giving us an insight for the meaning of the expression $x_{n+1} = Base_n + n\,(Base_n - Base_{n-1})$. Call the difference between two consecutive bases $\Delta_n = Base_n - Base_{n-1}$, and we get $x_{n+1} = Base_n + n\Delta_n$. So the value of x_{n+1} depends on the $Base_n$ and on the difference Δ_n between two consecutive bases. Also recall $M_n = \frac{1}{|Centre_n|}\sum_{k=1}^{|Centre_n|}\tilde{x}_k = Base_n$, is increasing. This implies that the falls in the X_n are due to a small distance Δ_n between the medians (i.e. between the points determining them in the sorted sequence). In order to show that a maximal element exists will show that the centre never reaches some element on the right-hand side in the ordered sequence. This would implies that since the distances between the elements are limited the newly generated x_n are bounded.

Suppose that X_n has $n < \infty$ elements and $m \leq n$, $t = 1 \ldots n$, then $\exists x_m \in X_n$ s.t. $x_m \geq x_t$. Then $\tilde{x}_n = x_m$ will be the rightest element in the sorted sequence. Since by corollary 15 the centre is traveling to the right with half of the speed of adding of new elements to the sequence, for the centre to reach the $\tilde{x}_n$, the sequence must add L new elements to the right of $\tilde{x}_n$ where

$$L \geq 2 \times \frac{n}{2} + 2 \times r = n + 2 \times r, \tag{0.9}$$

where r is the number of elements added to the left of $\tilde{x}_n$ until the centre reaches $\tilde{x}_n$. It is therefore possible for the centre to never reach $\tilde{x}_n$. Since $\tilde{x}_n$ is the rightest, the next $Base_{n+1}$ (i.e. M_{n+1}) produced from X_n^{Sort} will not use it, except if $n = 1\ or\ 2$ in which cases the sequence halts immediately and we are done by corollary 11. So the next median, i.e. base is produced from some smaller $\tilde{x}_p$ $(,\, \tilde{x}_{p+1})$ in X_n^{Sort}. So the base contribution for the new x_{n+1} will be smaller than the currently maximal element in the sequence. The second contribution factor is from the difference $\Delta_n = Base_n - Base_{n-1}$ between the current and the previous bases.

1. Suppose Δ_n is such that $x_{n+1} < x_n$, then the new x_{n+1} will be placed to the left of the maximal x_n in the sorted sequence X_{n+1}^{Sort} but to the right of the current median by lemma 16, so x_{n+1} will be positioned between some $\tilde{x}_q$ and $\tilde{x}_{q+1}$, where $q > p$ $(,\, p+1)$ and $q + 1 < n$. This also implies that the $r - count$ is increased by one and the distance between some two points will be split into two smaller distances. In detail: if the following x_{n+t}, $t > 1$ are such that equation 0.9 is:

1.1. never satisfied we are done since the maximal so far x_n will remain maximal.

1.2. enough x_{n+t} are generated such that x_{n+1} becomes part of the centre at some $l-th$ iteration in future. Since x_{n+1} is placed between some $\tilde{x}_q$ and $\tilde{x}_{q+1}$ then the distance between the latter will be cut in two pieces. Now:

1.2.1. if $\Delta(\tilde{x}_q,\ x_{n+1},\ \tilde{x}_{q+1})$ is less than the critical distance then the generated x_l will be positioned to the left of $\tilde{x}_{q+1}$ thus injecting a new element to the right of the $Base_l$ ($Base_l$ is determined by $\tilde{x}_q$ and x_{n+1}), and since it is injected to the left of x_n it increases r in equation 0.9 by at least two (one for x_{n+1} and one for x_l). Also, the distance between of x_{n+1} and $\tilde{x}_{q+1}$ is now again cut in two parts by x_l, thus further decreasing Δ_l because clearly $\frac{(n+1)\Delta}{2} \ll n\Delta$.

1.2.2. if $\Delta(\tilde{x}_q,\ x_{n+1},\ \tilde{x}_{q+1})$ is larger than the critical distance then the generated x_l will be positioned to the right of $\tilde{x}_{q+1}$, so:

1.2.2.1. x_l may be the new largest value - in this case it will be ignored for at least $L - many$ new elements by equation 0.9, again also increasing the $r - count$ by one.

1.2.2.2. x_l will be positioned between $\tilde{x}_{q+1}$and $\tilde{x}_{q+2}$ thus increasing the $r - count$ by least two (one for x_{n+1} and one for x_l) and additionally cutting the distance between $\tilde{x}_{q+1}$and $\tilde{x}_{q+2}$ in two parts, as in 1.2.1.

So if $x_{n+1} < x_n$, then in every case r increases by at least one, so at least two more elements are need to add to the sequence (in addition to the already required number) on the right hand side of x_n (i.e. larger than x_n values) before the centre could reach x_n, while at the same time the distance between two points have been split into two distances and thus making the advances of two x_l, x_{l+1}smaller instead of having one larger. So $x_{n+1} < x_n$ stabilizes x_n as a maximum.

2. Suppose that $x_{n+1} > x_n$, so now x_{n+1} is the new maximal value. But it is not possible to have continuously new largest values added to the sequence because largest value is determined by large base and large Δ_n, but the largest x_n are ignored (because of sorting) for at least L new elements, so some smaller values are used for base and Δ_n. The newly generated values, as shown in (1) above, are then injected somewhere in the sorted sequence to the right of the (base) median value used in their computation and to the left of the maximal values, further introducing smaller bases and dividing any larger distance.

3. Suppose that $x_{n+1} = x_n$ then:

3.1. if the centre never reach the $x_{n+1} = x_n$ as placed in the sorted sequence we are done;

3.2. if the centre reach $x_{n+1} = x_n$ as placed in the sorted sequence then the sequence halts by theorem 10 part a) and we are done.

Since the median travels 1/2 of the speed of adding new values, and for every new x_{n+1} added to the sequence smaller than the maximal:

a) two new values larger than the maximal will be needed to overtake it;

b) x_{n+1} divides the distance between the points between which it is injected in the sorted sequence and thus produce two new smaller x_{n+i} and x_{n+i+1} since $\frac{(n+1)\Delta}{2} \ll n\Delta$ instead of just one larger x_{n+i}. These two new smaller x_{n+i} and x_{n+i+1} promote further division by the same reason. Further for each smaller x_{n+i} two larger than the maximal values are required by a) to compensate for it, which become increasingly difficult since the interval bcomes denser and denser populated, with smaller and smaller distances between the elements.

So we have a continuously becoming denser area of elements, and for some large enough $\tilde{x}_n$, and large enough n, but finite, n-times the distance $n\Delta_n$ between any two M_n and M_{n-1} is insufficient to extend the base M_n for the next x_{n+1} beyond $\tilde{x}_n$, and thus $\tilde{x}_n$ becomes, (i.e. remains) the maximal element $x_{maximal}$. □

Remark. Again, briefly when and why X_n collapses: looking into figure 4 above clearly X_n is non-monotone. The behaviour of X_n is determined by the equation $x_{n+1} = Base_n + n\,(Base_n - Base_{n-1})$, where $Base_n \equiv M_n$. Thus $Base_n$ is determined by C_n and $Base_{n-1}$ by C_{n-1}, so x_{n+1} by three consecutive points in the middle of the sorted sequence. When it so happens that the distance between the upper two points is smaller than the distance between the lower two points X_n drops, that is $x_{n+1}<x_n$. The exact drop depends mostly on the values of the three points and to a lesser degree to how large n is.

Corollary 20. *The sequense of medians (M_n) is bounded above by $x_{maximal}$.*

Proof. Follows immediately from lemma 16 and theorem 19. □

Corollary 21. *The sequence $X_n \to (M_n)$ as $n \to \infty$.*

Proof. Let X_n be a recursive sequence generated by the mean median map from starting set $\{x_1,\, x_2, \ldots,\, x_k\}$ and suppose $n > k+1$, then recall $x_{n+1} = M_n + n\,(M_n - M_{n-1})$ is true by theorem 6. Since (M_n) is increasing by theorem 4 and (M_n) is bounded above by $x_{maximal}$ then $(M_n - M_{n-1}) \to 0$, so $x_{n+1} \to M_n$ as $n \to \infty$. □

Theorem 22. *There exists least upper bound $x^*_{maximal}$ for (M_n) such that $x^*_{maximal} \leq x_{maximal}$.*

Proof. Suppose X_n contains enough elements so that n-times the distance $n\Delta_n$ between any two M_n and M_{n-1} is insufficient to extend the base M_n for the next x_{n+1} beyond $\tilde{x}_n$, so $\tilde{x}_n = x_{maximal} = x^*_{maximal}$. But since the maximal is achieved all new elements added to the sequence are added to the left of $\tilde{x}_n$ thus increasing the density of elements between the centre and $\tilde{x}_n$. When sufficient density is achieved the $\tilde{x}_{n-1}$ element in the sorted sequence becomes unreachable for the same reasons as $\tilde{x}_n$ before it, so $\tilde{x}_{n-1}$ becomes the new least upper bound, so $x^*_{maximal} = \tilde{x}_{n-1} \leq \tilde{x}_n = x_{maximal}$. The process of lowering the least upper bound continues:

1. If the sequence halts until the sequence halts in which case $x^*_{maximal} = x_{halted}$.

2. If the sequence does not halt $x^*_{maximal} \to M_n$ as $n \to \infty$. □

Remark. Effectively, these are sequences attracting/pushing onto each other, on one hand M_n is increasing (though bounded above) pushing upwards, and on the other hand X_n which is overall decreasing pushing down towards M_n.

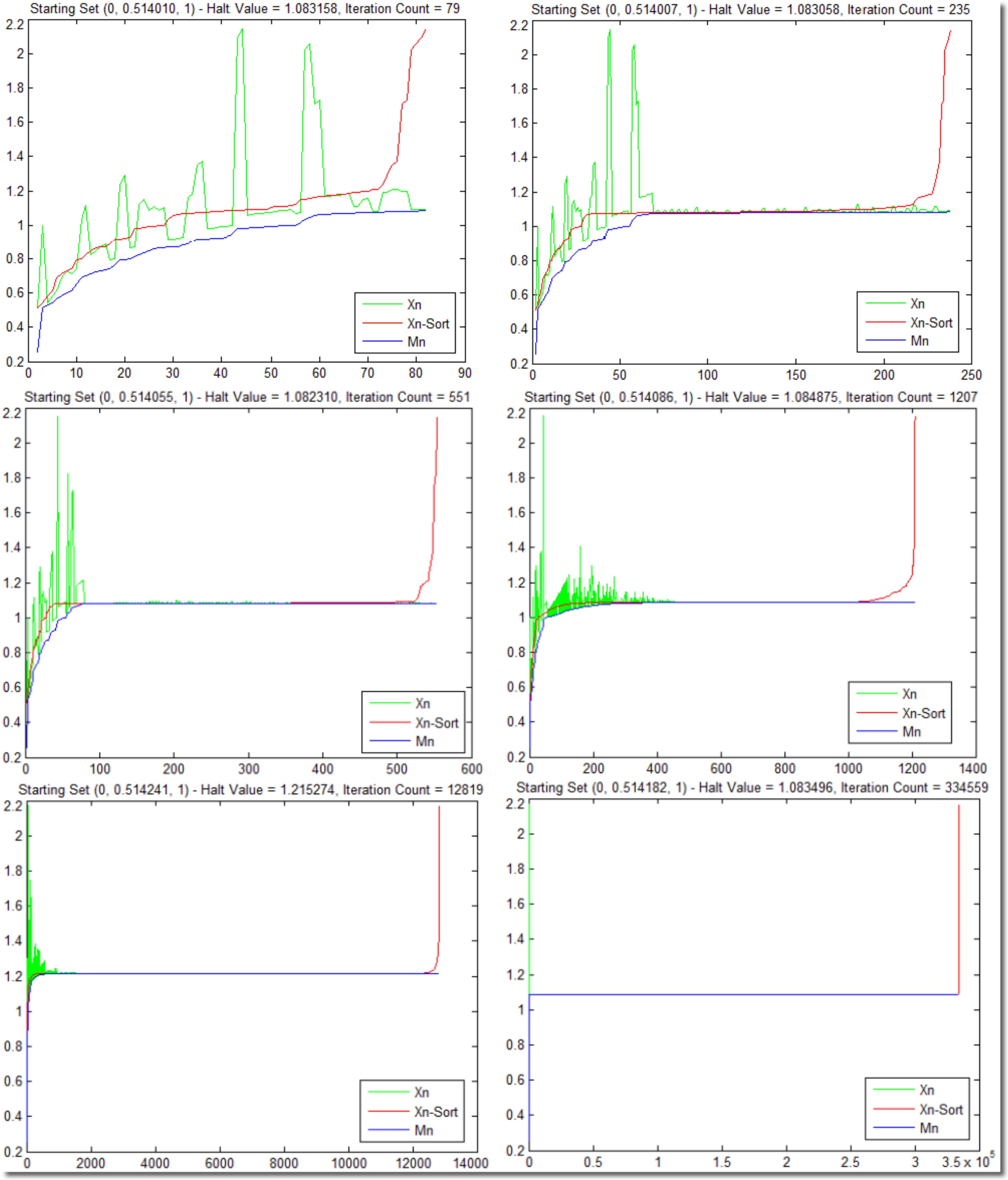
Theorem 23. *X_n and M_n are Cauchy sequences.*

Proof. If the sequence halts after finitely many iterations then the theorem is immediately true for both X_n and M_n. If the sequence does not halt then M_n is a Cauchy sequence immediately from theorem 4 and corollary 20. Then by corollary 21 X_n is also a Cauchy sequence. □

Proposition 24. *The weak terminating conjecture is true.*

Proof. Follows immediately from corollary 20. □

Figure 7. Plot of sequences until they halt. All over 20,000 experiments yield similar results.



Starting Set (0, 0.514010, 1) - Halt Value = 1.083158, Iteration Count = 79

Starting Set (0, 0.514007, 1) - Halt Value = 1.083058, Iteration Count = 235

Starting Set (0, 0.514055, 1) - Halt Value = 1.082310, Iteration Count = 551

Starting Set (0, 0.514086, 1) - Halt Value = 1.084875, Iteration Count = 1207

Starting Set (0, 0.514241, 1) - Halt Value = 1.215274, Iteration Count = 12819

Starting Set (0, 0.514182, 1) - Halt Value = 1.083496, Iteration Count = 334559

Remark. Although it may seem that the sequences have the same maximal value they in fact do not have the same maximal value. Refer to figure 3 where is a plot of the sequence vs. the max value for 10000 sequences on the interval 0.514 - 0.515.

Remark. Some halt kernels formed by the mean median map could be void, that is halt kernels with a value larger than the least upper bound of the (M_n) sequence will be never reached and absorbed by the centre and thus could never halt the sequence, and so are void.

Lemma 25. *Let $n > 2$ and n-even, then $M_{n+1} - E_{n+1} = M_n - E_n$, and for all $n > 0$ and n-even, then $M_{n+1} - M_n = M_n - M_{n-1}$.*

Proof. By theorem 7, $M_{n+1} - E_{n+1} = M_n - E_n$ is the same as $M_{n+1} - M_n = M_n - M_{n-1}$. Now, n is even so $n \pm 1$ is odd, so:

$$M_{n+1} - M_n = x_{\frac{n+2}{2}} - \tfrac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) = \tfrac{1}{2}\left[x_{\frac{n}{2}+1} - x_{\frac{n}{2}}\right]$$

$$M_n - M_{n-1} = \tfrac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) - x_{\frac{(n-1)+1}{2}} = \tfrac{1}{2}\left[x_{\frac{n}{2}+1} - x_{\frac{n}{2}}\right] \qquad \square$$

Conclusion. The author believes that Strong Terminating Conjecture is true. The reasons for this belief are:

Clearly all points in the X_n and M_n sequences are linearly dependent. For example, consider the sequence with starting set {0, 0.7, 1}. As we saw previously it halts in 12 iterations, in fact in 9 actual as the first three are building the starting set. Expressing the two x_n values from the halt kernel by unwinding the recursive iterations that produced them we have:

$$x_5 = M_4 + 4\left(M_4 - M_3\right) = \tfrac{1}{2}\left(\tilde{x}_3 + \tilde{x}_2\right) + 4\left(\tfrac{1}{2}\left(\tilde{x}_3 + \tilde{x}_2\right) - \tilde{x}_2\right) = \tfrac{5}{2}\tilde{x}_3 - \tfrac{3}{2}\tilde{x}_2 = 1.45$$

$$x_8 = M_7 + 7\left(M_7 - M_6\right) = \tilde{x}_4 + 7\left(\tilde{x}_4 - \tfrac{1}{2}\left(\tilde{x}_4 + \tilde{x}_3\right)\right) = \tfrac{1}{2}\left(9\tilde{x}_4 - 7\tilde{x}_3\right) = \tfrac{1}{2}\left(9\left[4M_3 - S_3\right] - 7\tilde{x}_3\right) = \tfrac{1}{2}\left(9\left[4\tilde{x}_2 - \tilde{x}_3 - \tilde{x}_2 - \tilde{x}_1\right] - 7\tilde{x}_3\right) =$$
$$-8\tilde{x}_3 + \tfrac{27}{2}\tilde{x}_2 = 1.45$$

In addition to this observation we also observe that for every even $n = 2k$, $k \in \mathbb{N}$, the median definition divides the sum between two elements of X_n: $M_{n=2k} = \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$. On the other hand for every even n $x_{n+1} = M_n + n\left(M_n - M_{n-1}\right) = M_n + 2k\left(M_n - M_{n-1}\right)$ (theorem 6) cancels that division. Also we note that this equation is symmetric to the median definition for even $n : M_{n=2k} = \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}) = x_{\frac{n}{2}} + \frac{1}{2}(x_{\frac{n}{2}} - x_{\frac{n}{2}+1})$. Since all points in X_n and M_n are linearly dependent, and since X_n and (M_n) are Cauchy sequences the X_n and (M_n) points get arbitrarily dence as $n \to \infty$, as well as $Q = (q_n)$, with $Q_n = (x_1, M_1, x_2, M_2, x_3, M_3, \ldots, x_n, M_n)$ by corollary 21 and theorem 23. So we have arbitrarily dense sequences of linearly dependent points, where X_n (and Q_n) are NOT monotone. So it is plausible to expect that somewhere in the finite future an already generated value will emerge again and thus create a halt kernel, repeating the process until a non-void halting kernel halt the sequence.

A few final thoughts that might be helpful:

1) We have an arbitrarily dense sequences of linearly dependent points, where X_n is NOT monotone. Also we have the sequence halting for at least one starting set, say {0, 0.7, 1}, so if the median map represents a complete class then it halts for all starting sets.

2) Periodic functions. Fourier analysis and convolution.

3) Chinese reminder theorem.

This project being a MSci thesis is time and resource constrained. Since the author has number of other modules and obligations requiring his attention, regretfully he was unable to make a serious attempt to formalize the above conclusion and transform it into a proof.

Part IV

Software

4.1 Object Specialization Model

We consider software modelling in object oriented environment, using C++ in particular.

It is not unusual for people to wonder why in the law of God lies are forbidden. One possible answer is that in an absolute world lies cannot exist or if they exist in a world then that world is not absolute. This is so because for a lie to exist it requires another lie to maintain it and thus if a lie is introduced then eventually the world becomes entirely self-contradictory. Hence in the world of God, which by definition is complete and omniscient a lie is impossible. Take this affirmation and translate is to the relation between a world that has been modelled by software and the world of the software itself. The software must represent the universe of discourse precisely or if this fails to be the case, then early or late there will appear one or more fundamental contradiction in it, which will make the software eventually impossible to exist (live) - grow. This implies necessity for the software model to be in one to one correspondence with the world that it describes, i.e. is to have no "lies" in it.

One of the implications of the above proposition is that names of variables must be precise. Comprehensive namespace however is only possible within comprehensive model. On the other hand comprehensive model is not possible without comprehensive namespace.

Suppose all variables in some software are named precisely with the names of entities that they precisely represent. What makes it impossible to use two or more variables of the same type interchangeably but inappropriately? Suppose that there are two Boolean variables representing the values of two unrelated propositions say: "Today is Monday" and "Water flows". Clearly they have nothing in common and must not be confused or mixed, however since the two variables representing the propositions are of the same type it is possible to erroneously assign the value of one of them to the other or use them in the same Boolean expression inappropriately, such as using the incorrect variable in a function call or other. So although the variable names represent some entity precisely errors are possible. These errors if misunderstood and corrected erroneously will lead to failure to maintain bijective representation of the universe of discourse.

Thus we identified the existence of another dimension to each and every variable which is its context. The author hereby proposes the idea to add a new dimension to objects, which would accommodate the information for context of the variable. This information then can be used by the compiler to ensure that the variable is being used only in the correct context, and if used improperly raise an appropriate error. Variables from different contexts could be used cross-context only via appropriate explicit context conversion.

The property maintaining the context information must be part of the meta-space, as opposed to part of the object. Thus the context of the object does not in any way alter the physical footstep of the object or its performance. All context information, checks, conversion and other properties and information are maintained explicitly in the meta-space and have no impact whatsoever on the compiled software and its performance.

In fact context already exist when considering types. The context of a type is defined by the namespace (if any) where the type is defined or the container (if nested), its ancestors and the overload constructors, methods and conversion operators. Presently context of variables also exist but only at physical level, which can be declaration scope (stack frame), containing class (object) or both. This is why two variables of the same type existing in compatible stack frames can be freely misused. The proposal in this paper is to add logical scope to variables, i.e.

context of objects, and thus prevent misuse in the same way as object from incompatible types cannot be misused without explicit conversion.

Adding context to a variable must effectively modify the methods/conversion operators of the class of the variable in a way so that there is a consistent behaviour. Ideally there would be native compiler support, where appropriate syntax might be for example:

```
bool<~ ContextA ~> bObjectName1;
bool<~ ContextB ~> bObjectName2;
```

It is possible to achieve variable specialization, i.e. attach context to variables with the facilities that C++ already have, although dedicated native support would be preferable. By using template specialization and inheritance we are able to achieve the required results. Consider the template class definition:

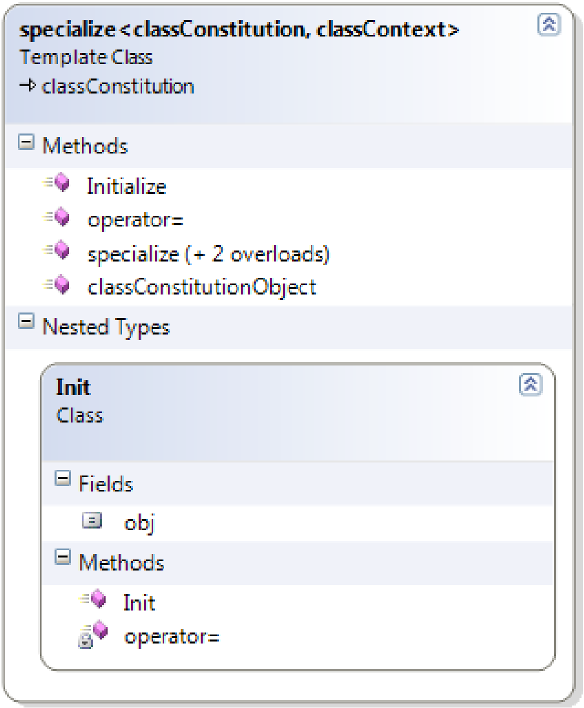
Listing 4. Template class achieving variable specialization using already available C++ facilities.

```
template< class classConstitution , class classContext > class specialize : public classConstitution
{
public:
   class Init
   {
   public:
      const classConstitution& obj;

   public:
      explicit Init( const typename classConstitution& obj ) : obj( obj ) {}

   private:
      Init& operator=( const Init& objInit )
      {
         obj = objInit;

         return( *this );
      }
   };

   static Init Initialize( const typename classConstitution& obj )
   {
      return( Init( obj ) );
   }

   specialize()
   {
   }

   specialize( const Init& obj ) : classConstitution( obj.obj )
   {
   }

   specialize( typename const specialize< classConstitution , classContext >& obj ) : classConstitution( *(classConstitution*)&obj
        )
   {
   }

   typename specialize< classConstitution , classContext > operator=( typename const specialize< classConstitution , classContext >&
         obj )
   {
      __super::operator=( obj );

      return( *this );
   }

   // prohibited to  prevent unwanted assignments.
   // typename specialize< classConstitution , classContext > operator=( typename classConstitution& obj ) { ... }
} };

#define as ,
#define by ,
#define of ,
```

An object from this type specialized with a type that we want to be intrinsic for the object and a second type giving the context is in fact a variable of the intrinsic type specialized with the context type, which is our objective. An object defined and specialized with the help of this template is less attractive than native implementation would be but still gives the required results:

```
specialize< bool , ContextA > bObjectName1;
specialize< bool , ContextB > bObjectName2;
```

Figure 8. Class diagram of the specialize< parameters > template.



In the template definition above we only allow one directional assignment, which may seem overly strict but as we shall see it is necessary. The problem may seem to be that assignment such as: bObjectName1 = true/false; generates error since the compiler does not know how to assign the reference-able (R) value to the locatable (L) value. Adding the banned operator= resolves this issue (in fact not an issue); however this clears the path for assignments such as: bObjectName1 = bObjectName2;.

This would defeat the purpose of the additional abstraction that we aim to introduce. Through the implied conversion to the ancestor type allowed by the public visibility of the superclass, and then through the overloaded assignment operator of the specialiser class the compiler quietly assigns incompatible-by-specialization (but compatible by type) variables, through the common superclass. Thus to break the chain we either have to disable the implied conversion or to remove the compatible function overload, or both. When working with existing code there would be many instances of objects from the superclass already existing in the code as well as functions with parameters of the superclass. Prohibiting the implicit conversion from specialized to non-specialized objects (i.e. to the superclass) would lead to cumbersome programming due the necessity for explicit conversion for every assignment or use of specialized type where non-specialized is expected. On the other hand it is most important to ensure that invalid assignment is impossible. Thus the solution is to maintain public inheritance of the superclass and prohibit all compatible operators and methods. We therefore remove the compatible operator overload and constructor which ensures an error in the assignment: bObjectName1 = bObjectName2;.

The removal of the compatible assignment operator however is the reason for impossibility of an assignment of object of the superclass:

```
bObjectName1 = true;  // generates error
bObjectName2 = false; // generates error
```

We cannot introduce a constructor with appropriate type to initialize the specializer as this will lead to the same problem as with the dedicated operator=, namely bypassing the context-wall which we aim to create. This we introduce the auxiliary type Init and helper function Initialize(...) to help the initialization:

```
bObjectName1 = specialize< Bool, ContextA >::Init( true );
bObjectName2 = specialize< Bool, ContextB >::Init( false );

// or

bObjectName1 = specialize< Bool, ContextA >::Initialize( true );
bObjectName2 = specialize< Bool, ContextB >::Initialize( false );
```

Remark. We do not make distinction between built-in type and user-defined type. For the purposes of this proposal we shall supply a wrapper class to any used build-in type in order to maintain consistency.

Definition 26. Context Specialized Type or just Context Type is a type composed by two or more types (possibly context specialized) and is constructed in the metaspace explicitly when declaring an object thus defining its type. The first of these types called constitution-class (or superclass, or type-class), determines the character and behaviour of the object, thus the object "is of" that type. The remaining one or more types called context-class(es) determine the relationships which the object could have, i.e. its context.

Example. Using classes bool and Colour we define specialized types:

```
bool<~ Color ~> boolMyPulloverBlue;
Color<~ bool ~> argbColorSuitable;
```

Using the above template implementation:

```
specialize< Bool, ContextA > bObjectName1( specialize< Bool, ContextA >::Initialize( true ) );
specialize< Bool, specialize< ContextA, ContextB > > bObjectName1( specialize< Bool, specialize< ContextA, ContextB > >::
      Initialize( true ) );
```

Remark. The definition of Context Specialized Type does not in any way restrict the complexity of the constitution and context classes.

Consider the task to place the type of a file in a string. Some software architects may take the approach to specialize a type i.e. derive a type e.g.:

```
class FileTypeDescription : public string
{
   // do almost nothing here, most work is in the base class
};
```

In practice almost all architects will simply use the string class just as it is. To use the context specialization methodology however we can create an auxiliary type with an absolutely empty content e.g.:

```
namespace sX
{
   class FileTypeDescription { /*nothing*/ };
};
```

Then simply context specialize the string object with it, achieving the desired result:

```
specialize<string,sX::FileTypeDescription> strType(specialize<string,sX::FileTypeDescription>::Initialize(TEXT("File␣type")));
```

To improve readability we define pre-processor syntax definitions for appropriate prepositions such as "as" and "by" with coma, thus from now on we shall use the appropriate preposition instead of coma in order to improve readability. It is a good practice to declare empty context-classes declared for the sole purpose to help constituting of Context Specialized Types in a dedicated namespace, thus separating them from the rest of the model. By convention this name space is called sX. Clearly using auxiliary empty context-classes is wrong when there are proper non-sX classes available in the model, which could be used as context-classes, or when an appropriate context could be assembled from them. For example suppose that the non sX-classes FileType and Description exist. Then we could do the same declaration as above as follows:

```
specialize< string as specialize< FileType by Description > > strType;
```

If there are classes File, Type and Description existing in the model, then we could construct the context:

```
specialize< string as specialize< Description of specialize< Type of File > > > strType;
```

In a comprehensive model most classes would exist as non sX, and very few sX classes will be required. When using multi-layer specialization as in the above two examples there may rise some disputes as to which class should be constitution-class and which context-class, e.g. should we have specialize< FileType, Description > or specialize< Description, FileType >. Consider the following prototype:

```
Pair< specialize< Integer as Quotient >, specialize< Integer as Reminder > > Integer::operator /( Integer iDividend, Integer
    iDivisor );
```

Now one need to define classes: class Quotient { ... } and class Reminder { ... } for this declaration to be meaningful. On the other hand Quotient and Reminder are integers, so there is a legitimate question whether defining class Quotient : public Integer { }; and class Reminder : public Integer {};, or even better template< class T> class Quotient : public T {}; and template< class T> class Reminder : public T {}; is not better than using the Object Specialization Model? Defining Quotient and reminder as templates may seem similar to the original declaration, but they are clearly very different: specialize< Integer as Quotient > specializes an Integer variable as quotient, that is limits its external friends and field of connections. The other however specializes the internal nature of the type Quotient. The alternative to the original declaration is then: inline Pair< Quotient< Integer >, Reminder< Integer > > operator /(Integer iDividend, Integer iDivisor);. At first glance this declaration will have immediate consequences similar to the original one, provided that casting operators and implicit constructors are unavailable, but in fact this is a wrong way to go. The reason is that the type Quotient is not restricted to only the whole part of division, but in a larger generalization (besides general homonyms). Quotient can also be a set, group, space, etc, and template specialization cannot truncate this larger generalized type to the small reminder from division case. Further Quotient< double > is a self-contradictory statement however there is no intrinsic way to prohibit such declaration, while Reminder< double > is perfectly all right. Quotient and Reminder are abstract entities and when defined as types required careful attention. Thus the specializer approach above is the correct one in this case. The analysis of this model could be further extended but will be omitted here. The most important consideration as always is to maintain bijective relation between the software model and the Universe of Discourse.

Suppose there is an enumerator defined as:

```
enum Result
{
   Success = 0,
   Failed = 1,
   Exception = 3
   // ...
};
```

This scenario is not different than an integer int< Width > when there is a native support for the Object Specialization Model.

```
Result<~ sX::Item ~> resultItem1;
Result<~ sX::Item ~> resultItem2;
```

However if we are using the substitute methodology we will have to define a class for each enumerator, which would include the native enum and operate on its behalf. Use of template specialized with the enumerator or a declaration map propagating the native enumerator might be useful to simplify the task in a generic way.

The Object Specialization Model brings several advantages to software that implements it:

1. Safer code – the object specialization model prevents use of incorrect variables when they are from the same type but different contexts. Consider the following example form a commercial application:

```
bool Save( const bool bKeepOnTopC,
           const bool bKeepOnTopO,
           const bool bUseFullPath,
           const bool bPreviewFile,
           const bool bShufflePlay,
           const bool bPlayNextDir,
           const bool bLoopDirDirs ) const throw();
```

Such functions with large list of parameters are not unusual in professional software development. Regardless whether this large list of parameters is due to bad design or is indeed intrinsic for the function, the possibility for an error using the incorrect Boolean value is there. To prevent from errors the software engineers are required to invest extra effort. Using the object specialization model we transform the above function to a safe one as follows:

```
bool Save( const specialize< Bool, sX::OnTopCompDlg > bKeepOnTopC,
           const specialize< Bool, sX::OnTopOpenDlg > bKeepOnTopO,
           const specialize< Bool, sX::UseFullPath  > bUseFullPath,
           const specialize< Bool, sX::PreviewFile  > bPreviewFile,
           const specialize< Bool, sX::ShufflePlay  > bShufflePlay,
           const specialize< Bool, sX::PlayNextDir  > bPlayNextDir,
           const specialize< Bool, sX::LoopDirDirs  > bLoopDirDirs ) const throw();
```

In the next example we will protect the FILETIME members of a structure from the same type of error.

```
struct FileData
{
   unsigned __int64 uiFileSize;
   DWORD dwAttributes;

   specialize< FILETIME as sX::CreationTime    > ftCreationTime;
   specialize< FILETIME as sX::LastAccessTime > ftLastAccessTime;
   specialize< FILETIME as sX::LastWriteTime  > ftLastWriteTime;

   string strFilepath;
};
```

2. Descriptive self-explanatory code – the definition of a context specialized object it is not only the name of the object but also the Context Type which carries additional information for the meaning and purpose of the variable. Thus the code becomes much more readable and meaningful. In addition the type information, i.e. the constitution and context classes' information is also displayed by the Intelli-sense of the IDE and also in the watch windows of the debugger making writing and debugging of code more efficient.

3. Expanding and structuring the namespace/type-space - this is an architectural benefit for the particular software model being developed. Because of the context specialization, software architects now have more points and relationships to base their models on and thus should have less number of failures in their attempts to achieve bijective model. Generally, models always fail to be bijective and that is why fresh new versions of software are developed. Consider the multibillion investments in Windows XP and the complete rewrite done for Windows 7.

4. Language function space expansion. Suppose the class:

```
class MyClass : public MyClassParent
{
   // ...
   virtual bool operator == ( const string& strFileNameToCompare ) const;
};
```

is well defined and working on a whole hierarchy of types, with multiple instantiations and calls to the above comparison operator. The passed as parameter string is interpreted as a filename and is used to compare the content of the file with the content of the particular object invoking the method. Suppose that later one needs to have another operator with the same prototype:

```
virtual bool operator == ( const string& strStringToCompare ) const;
```

but this time they want to compare the actual string with the content of the object. However since the prototype is already used the new function cannot be implemented with the same prototype. Traditionally the solution would be to add another parameter, thus changing the prototype, and clarifying to the compiler which function we would like it to call. It is clear that such an approach is crude, but in this particular case it is not even possible since operator== can have only one parameter. There are also other possible crude solutions, requiring flags, initializations, definition of special unwanted types, etc. The Object Specialization Model however presents quick and elegant solution. The solution of this problem using the suggested methodology is to specialize the parameter of the function as follows:

```
class MyClass : public MyClassParent
{
   // ...

   // original declaration – could be also kept if wanted.
   // virtual bool operator == ( const string& strFileNameToCompare ) const;

   // modified original declaration.
   virtual bool operator == ( const specialize< string as sX::FileName >& strFileNameToCompare ) const;

   // The newly added function.
   virtual bool operator == ( const specialize< string as string >& strStringToCompare ) const;
};
```

Now, should one desire they could specialize the parameter as they wish, including multi-level (nested) specialization if the universe is discourse that they work with requires it.

5. Context overloading – Inspired by the above methodology the Object Specialization Model allows a new third way of method overloading. The two other methods are:

a) Overwriting – when using polymorphic functions with the same prototype;

b) Overloading – when using functions with the same type but different parameter list.

And now the third way:

c) Context overloading – works on both polymorphic and non-polymorphic functions with the same name and the same parameter list where only the context-class of one or more parameters is different than that of any other overload. Example: the following three context-overloads are well defined overloads which will be distinguished by the compiler and called as appropriate according to the type of the second parameter of the call.

```
void MyClass::MyFunction( int , specialize< string , sX::FileName >& );
void MyClass::MyFunction( int , specialize< string , sX::FolderName >& );
void MyClass::MyFunction( int , specialize< string , string >& );
```

We finish the Object Specialization Model with a note that in some cases specialization may be needed on a verb as opposed to a noun. That is the constitution-class represents a verb or other type of entity instead of a noun as usual. The same is also true for the context-class. Given the ability to construct multi-layer (nested) Specialization Types this implies that an object could have Specialization Type which (literally) (re)presents a complete or partial sentence and if necessary even a paragraph, chapter or novel.

The Object Specialization Model is available from `http://www.MBBSoftware.com/Software/ObjectSpecializationModel/Default.aspx` under the MIT Open Source Software License Agreement. Future updates and development on the Object Specialization Model will be published at that address. The source code that follows is used in the Mean Median Map and in the examples of the above description.

Listing 5. Implementation of the Object Specialization Model using templates and inheritance.

```
//
//
// Specialize.h: Type Specialization definitions as example of implementation of:
//
//                   Object Specialization Model - Implementation Example One
//
// © Copyright 2009 - 2011 by Miroslav Bonchev Bonchev. All rights reserved.
//
//
//*****************************************************************************************************


// Open Source License – The MIT License
//
//
// {your product} uses the: Object Specialization Model – Copyright © 2009 - 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
// associated  documentation files  (the "Software"),  to deal  in the Software without restriction,
// including  without  limitation the rights  to use,  copy,  modify,  merge,  publish,  distribute,
// sublicense,  and/or sell copies of the Software,  and to permit persons to  whom the  Software is
// furnished to do so, subject to the following conditions:
//
// The  above  copyright  notice  and  this  permission  notice  shall be  included in all copies or
// substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT
// NOT  LIMITED  TO  THE  WARRANTIES  OF  MERCHANTABILITY,  FITNESS  FOR  A  PARTICULAR  PURPOSE AND
// NONINFRINGEMENT.  IN NO EVENT SHALL  THE  AUTHORS  OR  COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
// DAMAGES OR OTHER LIABILITY,  WHETHER IN AN ACTION OF CONTRACT,  TORT OR OTHERWISE,  ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.


//___________________________________________________________________________________________________
// This software is OSI Certified Open Source Software.
// OSI Certified is a certification mark of the Open Source Initiative.


#pragma once


#include "Common.h"


#define as ,
#define by ,
#define of ,


template< class classConstitution, class classContext > class specialize : public classConstitution
{
public:
   class Init
   {
   public:
      const classConstitution& obj;

   public:
      explicit Init( const typename classConstitution& obj ) : obj( obj ) {}

   private:
      Init& operator=( const Init& objInit ) { obj = objInit; return( *this ); }
   };

   static Init Initialize( const typename classConstitution& obj )
   {
      return( Init( obj ) );
   }

   specialize() {}
   specialize( const Init& obj ) : classConstitution( obj.obj ) {}
   specialize( typename const specialize< classConstitution, classContext >& obj ) : classConstitution( *(classConstitution*)&obj ) {}

   typename specialize< classConstitution, classContext > operator=( typename const specialize< classConstitution, classContext >& obj )
   {
      __super::operator=( obj );
      return( *this );
   }

   // Need to add the full spectrum of operators in future.
   typename specialize< classConstitution, classContext > operator!() const
   {
      return( specialize< classConstitution, classContext >( specialize< classConstitution, classContext >::Init( __super::operator!() ) ) );
   }

   typename classConstitution& GetConstitutionObject()
   {
      return( *this );
   }

   typename const classConstitution& GetConstitutionObject() const
   {
      return( *this );
   }
};


class Bool
{
private:
   bool b;

public:
   Bool() : b( false ) {}
   Bool( const bool b ) : b( b ) {}
   Bool( const Bool& b ) : b( b ) {}
   Bool( const int ib ) : b( 0 != ib ) {}
   ~Bool() {}

   Bool& operator=( const bool b ) { this->b = b; return( *this ); }
   Bool& operator=( const Bool b ) { this->b = b.b; return( *this ); }
   Bool& operator=( const int ib ) { this->b = 0 != ib; return( *this ); }

   bool operator==( const bool b ) const { return( this->b == b ); }
   bool operator!=( const bool b ) const { return( this->b != b ); }

   Bool operator==( const Bool b ) const { return( this->b == b.b ); }
   Bool operator!=( const Bool b ) const { return( this->b != b.b ); }

   Bool operator!() const { Bool b1( !b ); return( b1 ); }

   operator bool () const { return( b ); }
   operator bool& () { return( b ); }
};


namespace sX
{
   class FileTypeDescription {};
   class FileName {};
   class FolderName {};

   class Description {};
   class Type {};
   class File {};
   class Item {};

   class CreationTime {};
   class LastAccessTime {};
   class LastWriteTime {};

   class OnTopCompDlg {};
   class OnTopOpenDlg {};
   class UseFullPath  {};
   class PreviewFile  {};
   class ShufflePlay  {};
   class PlayNextDir  {};
   class LoopDirDirs  {};

   class Numerator {};
   class Denominator {};
   class Quotient {};
   class Reminder {};
};
```

4.2 Proper Numbers Library

The Proper Numbers Library is a small, efficient and fast set of C++ classes that allows representation of arbitrarily large integer numbers and arbitrarily precise rational numbers. It is easy to be integrated in any C++ project and used as if the classes defined in it are native for the language. The name "Proper" comes from the proper behaviour of the numbers that constitute it, and in particular the even distribution of rational numbers, and other "proper" features. In order to enforce stricter Object Oriented compliance some standard operators such as implicit conversion to Boolean values are not defined. For the needs of the Mean Median Map the classes need no complex mathematical functions such as sine, cosine, roots, logarithm, etc. thus there are not such functions presently defined. However, for the most common and usual uses the classes offer the required functionality and basic arithmetic operators. The library is published under the MIT Open Source Software License Agreement, and can be downloaded from `http://www.MBBSoftware.com/Software/ProperNumbersLibrary/Default.aspx`. Future updates and development on the Proper Numbers Library, such as adding more mathematical functions and additional types of numbers will be published at that address.

4.2.1. The Integer Class

The Integer class represents unsigned integers with arbitrary precision. Objects from this type expand arbitrarily large up to available memory and are able to accommodate arbitrarily large numbers (up to the available memory).

Integers in a digital computer are manipulated by bit fields called registers able to do bitwise operations and operations on the whole field, such as ADD (add), ADC (add with carry), SUB (subtract), AND (and), OR (or), etc. called instructions. There are number of flags in the processor's Control and Status register that signal or indicate about different conditions that may have occurred. For example overflow after an addition operation. Depending on the number of bits in the register it is a variable of type $\mathbb{Z}_{2^{number-of-bits}}$ usually $\mathbb{Z}_{2^8}$, $\mathbb{Z}_{2^{16}}$, $\mathbb{Z}_{2^{32}}$, $\mathbb{Z}_{2^{64}}$, etc. and operate in the respective modular arithmetic. High level languages utilize the processor registers to perform arithmetic and other operations on the content of the memory. Integer variables in high level languages are byte arrays with sufficient size located in the memory and are designated with its name in the metaspace. When an integer variable is larger than the largest register able to perform integer operations the compiler/assembly programmer uses typically a loop and the status flags applying the operation throughout the whole array working in $\mathbb{Z}_{2^{number-of-bits-in-variable}}$. Thus if a variable is 128 bit (16 bytes) and the largest arithmetic register is 16 bit, the compiler will place a function call for every arithmetic operation on the variable. In the body of function will be a loop from 1 to 16 iterations using appropriate processor instructions on only 16 bits (2 bytes) at a time achieving correct results in $\mathbb{Z}_{2^{128}}$.

The Integer class from the Proper Numbers Library uses similar approach but instead using constant size arrays to represents integers it uses a generic linked list. Each number is composited from one or more 64 bit words. Arithmetic operations are implemented using set of appropriate fast algorithms. When operation is about to produce a number with greater size then the current it is carried out adding more space to the number and thus overflows do not occur. Hence the integer class represents $\mathbb{N}$ (up to the available memory in the system). Shift to right and subtraction decrease the size of the number (object). Empty number is not allowed. Zero is represented with one 64 bit word set to zero. The Integer objects are maintained normalized which for this class means that there are no leading zeroed 64 bit words - except for the one zero word for the 0. When subtracting larger number from a smaller one the result wraps up and has the size of the larger number. The decrement operator has unusual behaviour - it does NOT decrement below zero to avoid ambiguity.

Remarkably, the Integer class is a specialization of a linked list of type MList$<$ unsigned __int64 $>$. When a number is represented with an object of this type the underlying linked list is accelerated thus the words of the number are accessed as an array achieving performance as if the numbers were indeed represented by an array.

The integer class inherits the list as protected access and thus the methods of the underling list are not visible (accessible) from an integer instance.

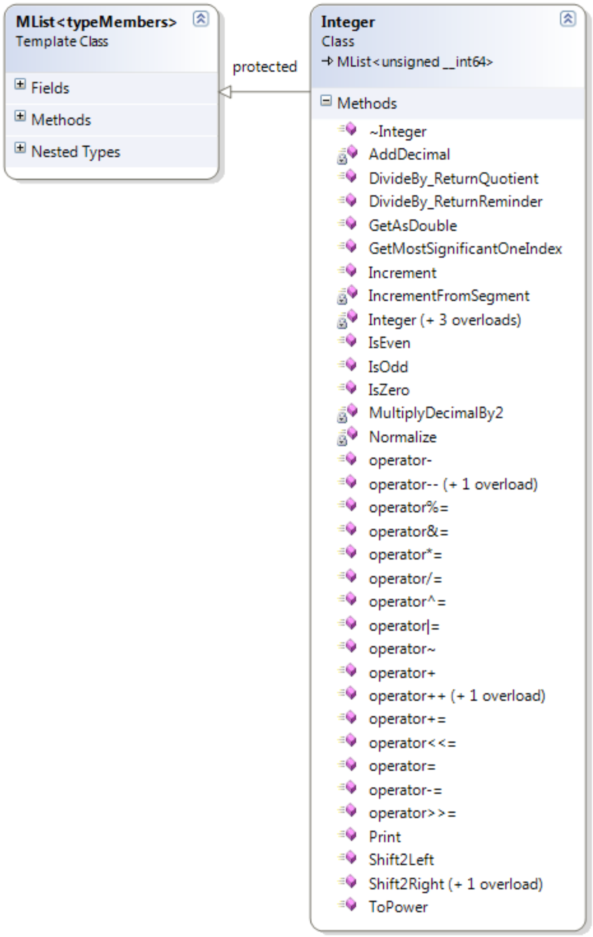
Object of type Integer throw NumberException only when dividing by zero. This behaviour may change in future removing all exceptions altogether although this is not likely, or new operator with alternative behaviour may be provided. The division operator returns a specialized as Quotient and Reminder pair thus helping to prevent errors.

The Boolean operators are NOT overloaded purposefully. The reason is that Integer numbers are NOT Boolean entities and therefore they should not implicitly represent Boolean values. Should one want to use them as Boolean variables they can use the IsZero() method and the comparison operators appropriately.

Objects from the Integer class are able to print themselves using the Print() method.

Remark. The Integer class represents ℕ and not ℤ purposefully. This is necessary to maintain correct model without redundancies. Signed integers $\mathbb{Z} \subset \mathbb{Q}$ can be represented using rational numbers with denominator 1. Otherwise there would be redundancies in the sign and computation of every rational number, which is a clear and definitive indication for inconsistent model.

Figure 9. Class diagram of the Integer class from the Proper Numbers Library.



Listing 6. The Integer class from the Proper Numbers Library.

```
//
//
// Integer.h: Unsigned integer with arbitrary precision.
//
//                Proper Numbers Library
//
// © Copyright 2011 - 2011 by Miroslav Bonchev Bonchev. All rights reserved.
//
//
//******************************************************************************************************


// Open Source License – The MIT License
//
//
// {your product} uses the: Proper Numbers Library – Copyright © 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
// {your product} uses the: Object Specialization Model – Copyright © 2009 - 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
// associated  documentation files  (the "Software"),  to deal  in the Software without restriction,
// including  without  limitation the rights  to use,  copy,  modify,  merge,  publish,  distribute,
// sublicense,  and/or sell copies of the Software,  and to permit persons to  whom the  Software is
// furnished to do so, subject to the following conditions:
//
// The  above  copyright  notice  and  this  permission  notice  shall be  included in all copies or
// substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT
// NOT  LIMITED  TO  THE  WARRANTIES  OF  MERCHANTABILITY,  FITNESS  FOR  A  PARTICULAR  PURPOSE AND
// NONINFRINGEMENT.  IN NO EVENT SHALL  THE  AUTHORS  OR  COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
// DAMAGES OR OTHER LIABILITY,  WHETHER IN AN ACTION OF CONTRACT,  TORT OR OTHERWISE,  ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.


//__________________________________________________________________________________________________
// This software is OSI Certified Open Source Software.
// OSI Certified is a certification mark of the Open Source Initiative.


#pragma once


#include "stdafx.h"
#include "Common.h"
#include "MAtom.h"
#include "StringEx.h"
#include "Pair.h"
#include "Specialize.h"


#pragma warning( push )
#pragma warning( disable : 4290 )   // warning C4290: C++ exception specification ignored except to indicate a function is not
__declspec(nothrow)

template< class dataType, bool bExceptions = true >
class NumberException
{
public:
   enum Error
   {
      eIntegerDivisionByZero,
      eNonCompatibleOperation
   };

private:
   Error eError;

private:
   NumberException( const Error excError )                    throw() : eError( excError ) {}

public:
   NumberException( const NumberException& objNumberException ) throw() : eError( objNumberException.eError ) {}
   ~NumberException() {}

   NumberException& operator=( const NumberException& objNumberException ) throw() { eError = objNumberException.eError; }

public:
   __forceinline static void TestDivisionByZero( typename const dataType muiDivisor ) throw( NumberException< dataType, bExceptions > )
   {
      if( 0 == muiDivisor )
      {
         MASSERT( FALSE );

         bExceptions ? throw( NumberException< dataType, true >( NumberException::eIntegerDivisionByZero ) ) : 0;
      }
   }

   __forceinline static void TestCompatibility( typename const dataType objLeft, typename const dataType objRight ) throw( NumberException<
dataType, bExceptions > )
   {
      if( !objLeft.IsCompatible( objRight ) )
      {
         MASSERT( FALSE );

         bExceptions ? throw( NumberException< dataType, true >( NumberException::eNonCompatibleOperation ) ) : 0;
      }
   }
};

#pragma warning( pop )



// The Integer class represents unsigned integer with arbitrary precision.
//
// Objects from this type expand arbitrarily large up to available memory and are able to accommodate
// arbitrarily large numbers (up to the available memory).
//
// Some points of interest:
//
// Each number is composited from one or more 64 bit words. Arithmetic operations are implemented using
// set of appropriate fast algorithms. When a number exceed the current size the operation is carried
// out adding more space to the number thus overflow and truncation are inapplicable and hence it is
// not possible to lose information for these reasons (provided there is enough memory in the system).
// Shift to right and subtraction decrease the size of the number (object). Empty number is not allowed.
// Zero is represented with one 64 bit word equal to zero. The objects are maintained normalized which
// for this class means that there are no leading zeroed 64 bit words - except one zero word for the 0.
// When subtracting larger number from smaller the result wraps and has the size of the larger number.
// The decrement operator has unusual behaviour - it does NOT decrement below zero to avoid ambiguity.
//
// Semantically the Integer class is a specialization of a linked list of type MList< unsigned __int64 >.
// When a number is represented with an object of this type the underlying linked list is accelerated thus
// the words of the number are accessed as an array achieving performance as if the numbers were indeed
// represented by an array. The integer class inherits as protected and thus the methods of the underling
// list are not visible (accessible) from an integer instance.
//
// Currently object from the Integer type throw NumberException only when dividing by zero. This behaviour
// may change in future removing all exceptions altogether although this is not likely, or new operator
// with alternative behaviour will be provided. The division operator returns object specialized Quotient
// and Reminder thus helping to prevent errors.
//
// The logical operators are NOT overloaded purposefully. The reason is that Integer numbers are NOT Boolean
// entities and therefore they should not implicitly represent logical values. Should one want to use them
// as Boolean variables they can use the IsZero() method and the comparison operators appropriately.
//
// Objects from the Integer class are able to print themselves using the Print() method.
class Integer : protected MList< unsigned __int64 >
{
   template< class classType, class classSpecializer > friend class specialize;

private:
   // Constructor for internal use creating skeleton of a number.
   Integer( const Integer& integer, const DWORD dwSegmentCount )
   {
      if( 0 != dwSegmentCount )
      {
         *this = integer;

         for( DWORD dwIndex = GetCount(); dwIndex < dwSegmentCount; dwIndex++ )
         {
            AddTail( new unsigned __int64( 0 ) );
         }
      }
   }


   // Removes any leading zeroes, but leave one last if the number is zero. "Empty" number is illegal.
   Integer Normalize()
   {
      while( (0 == *pTail->GetObject()) && (1 < GetCount()) )
      {
         delete RemoveTail();
      }

      if( IsEmpty() )
      {
         AddHead( new unsigned __int64( 0 ) );
      }

      return( *this );
   }


public:
   // Default constructor - initializes the object with zero.
   Integer() : MList< unsigned __int64 >( (unsigned __int64)0 )
   {
   }


   // Initializes the object with native (unsigned) integer.
   Integer( const unsigned __int64 uiInteger ) : MList< unsigned __int64 >( (unsigned __int64)uiInteger )
   {
   }


   // Copy constructor.
   Integer( const Integer& integer ) : MList< unsigned __int64 >( integer )
   {
   }


   // Inherits virtual destructor.
   ~Integer()
   {
   }


   // Standard operator equal.
   inline Integer& operator=( const Integer& integer )
   {
      return( dynamic_cast< MList< unsigned __int64 >* >( this )->operator=( *dynamic_cast< const MList< unsigned __int64 >* >( &integer ) ),
*this );
   }


   // Test if the number is zero.
   inline bool IsZero() const
   {
      return( (1 == GetCount()) && (0 == *(*this)[0]) );
   }


   // Test if the number is even.
   inline bool IsEven() const
   {
      return( 0 == (1 & *(*this)[0]) );
   }


   // Test if the number is odd.
   inline bool IsOdd() const
   {
      return( 1 == (1 & *(*this)[0]) );
   }


   // Standard operator +=
   inline Integer operator +=( const Integer& iSummand )
   {
      return( *this = ::operator +( *this, iSummand ) );
   }


   // Standard operator -=
   inline Integer operator -=( const Integer& iSubtrahend )
   {
      return( *this = ::operator -( *this, iSubtrahend ) );
   }
```

```
// Standard operator *=
inline Integer operator *=( const Integer& iFactor )
{
   return( *this = *this * iFactor );
}


// Standard operator /=
inline Integer operator /=( const Integer& iDivisor )
{
   return( *this = (*this / iDivisor).GetLabelRef() );
}


// Standard operator %=
inline Integer operator %=( const Integer& iDivisor )
{
   return( *this = (*this / iDivisor).GetValueRef() );
}


// Standard operator &=
inline Integer operator &=( const Integer& iGate )
{
   return( *this = *this & iGate );
}


// Standard operator |=
inline Integer operator |=( const Integer& iGate )
{
   return( *this = *this | iGate );
}


// Standard operator ^=
inline Integer operator ^=( const Integer& iGate )
{
   return( *this = *this ^ iGate );
}


// Standard operator <<=
inline Integer operator <<=( const Integer& iShiftCount )
{
   return( *this = *this << iShiftCount );
}


// Standard operator >>=
inline Integer operator >>=( const Integer& iShiftCount )
{
   return( *this = *this >> iShiftCount );
}


// Unary plus (returns the operand itself).
inline Integer operator+() const
{
   return( *this );
}


// Unary operator negation (two's complements – inverts the bits and adds one).
inline Integer operator-() const
{
   if( IsZero() )
   {
      return( *this  );
   }

   Integer i( *this );

   for( DWORD dwIndex = 0; dwIndex < i.GetCount(); dwIndex++ )
   {
      *i[dwIndex] = ~*i[dwIndex];
   }

   i.IncrementFromSegment( 0 );
   return( i );
}


// Operator bitwise NOT (inverts all bits).
inline Integer operator ~() const
{
   Integer i( *this );

   for( DWORD dwIndex = 0; dwIndex < i.GetCount(); dwIndex++ )
   {
      *i[dwIndex] = ~*i[dwIndex];
   }

   return( i );
}


// Returns an integer power of this integer.
inline Integer ToPower( const Integer& uiPower ) const
{
   Integer uiThis( 1 );

   for( Integer uiCount = 0; uiCount < uiPower; uiCount += 1 )
   {
      uiThis = uiThis * *this;
   }

   return( uiThis );
}


// Add one to the number - this instance.
inline Integer& Increment()
{
   IncrementFromSegment( 0 );

   return( *this );
}


// Prefix increment operator.
inline Integer& operator++()
{
   Increment();

   return( *this );
}


// Postfix increment operator.
inline Integer operator++( int )
{
   Integer uiThis( *this );

   Increment();

   return( uiThis );
}


// Prefix decrement operator. Note it does NOT decrement below zero - since the number is
// of an arbitrary span it is ambiguous how large should the number be when decrementing
// it from zero. However since it is an unsigned integer the number will not be decremented
// below the zero, thus any decrement call after the number is already zero will be ignored.
inline Integer& operator--()
{
   if( !IsZero() )
   {
      *this = *this - 1;
   }

   return( *this );
}


// Prefix decrement operator. Note it does NOT decrement below zero - since the number is
// of an arbitrary span it is ambiguous how large should the number be when decrementing
// it from zero. However since it is an unsigned integer the number will not be decremented
// below the zero, thus any decrement call after the number is already zero will be ignored.
inline Integer operator--( int )
```

```
{
   Integer uiThis( *this );

   if( !IsZero() )
   {
      *this = *this - 1;
   }

   return( uiThis );
}


// Return ONE based index of the most significant 1, therefore returning 0 implies
// that the number is zero. Divide this method’s return value mod 64 plus one to
// get the number of 64 bit words used in the representation of the number.
inline unsigned __int64 GetMostSignificantOneIndex() const
{
   const unsigned __int64 uiMSB( *GetTail() );

   for( unsigned __int64 uiWindow = ((unsigned __int64)1) << 63, uiIndex = 64; 0 != uiWindow; uiWindow >>= 1, uiIndex-- )
   {
      if( 0 != (uiMSB & uiWindow) )
      {
         return( (GetCount() - 1) * 8 * sizeof( unsigned __int64 ) + uiIndex );
      }
   }

   return( 0 );
}


// Return a string object containing the decimal representation of the number.
MStringEx< TCHAR > Print() const
{
   MList< BYTE > listResult( new BYTE( 0 ), NULL );
   MList< BYTE > list2Power( new BYTE( 1 ), NULL );

   // One based index of the most significant one - zero implies tho ones at all.
   unsigned __int64 uiMSBitIndex( GetMostSignificantOneIndex() );

   if( 0 < uiMSBitIndex )
   {
      unsigned __int64 uiCurrentIndex( 0 );

      // Head is LSB, Tail is MSB
      for( DWORD dwSegment = 0; dwSegment < GetCount(); dwSegment++ )
      {
         for( unsigned __int64 uiWindow = 1; 0 != uiWindow; uiCurrentIndex++, uiWindow <<= 1 )
         {
            if( 0 != (*(*this)[dwSegment] & uiWindow) )
            {
               AddDecimal( listResult, list2Power );
            }

            if( uiCurrentIndex >= uiMSBitIndex )
            {
               break;
            }

            MultiplyDecimalBy2( list2Power );
         }
      }
   }

   MStringEx< TCHAR > strNumber;
   for( DWORD dwIndex = 0; dwIndex < listResult.GetCount(); dwIndex++ )
   {
      strNumber = MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%d"), *listResult[dwIndex] ) + strNumber;
   }

   return( strNumber );
}


// Return the quotient part from the division of this number by another integer.
// This method uses the division operator. It is more efficient to use the division
// operator directly if you also need the reminder part from tis division.
inline Integer DivideBy_ReturnQuotient( const Integer& iDivisor ) const
{
   return( operator /( *this, iDivisor ).GetLabelRef() );
}
```

```
// Return the reminder part from the division of this number by another integer.
// This method uses the division operator. It is more efficient to use the division
// operator directly if you also need the quotient part from tis division.
inline Integer DivideBy_ReturnReminder( const Integer& iDivisor ) const
{
   return( operator /( *this, iDivisor ).GetValueRef() );
}


// Shift to left.
inline Integer& Shift2Left( const unsigned __int64 uiTimes )
{
   if( !IsZero() && (0 != uiTimes) )
   {
      const unsigned __int64 uiNewSegments( uiTimes / 64 );

      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
      // Add one empty segment at the head, i.e. least segnificant part, for every 64 bit shift to the left.
      for( Integer iSegment = 0; iSegment < uiNewSegments; iSegment++ )
      {
         AddHead( new unsigned __int64( 0 ) );
      }


      const BYTE b1BitIncShift( uiTimes % 64 );

      if( 0 != b1BitIncShift )
      {
         // Division by 64 cannot have remiander bigger than 63.
         MASSERT( b1BitIncShift < 64 );

         // Head - LSB, Tail - MSB.
         unsigned __int64 uiCarry( 0 );
         for( DWORD dwIndex = 0; dwIndex < GetCount(); dwIndex++ )
         {
            const unsigned __int64 bNewCarry( *(*this)[dwIndex] >> (64 - b1BitIncShift) );

            *(*this)[dwIndex] <<= b1BitIncShift;
            *(*this)[dwIndex] |=  uiCarry;

            uiCarry = bNewCarry;
         }

         if( 0 != uiCarry )
         {
            AddTail( new unsigned __int64( uiCarry ) );
         }
      }
   }

   return( *this );
}


// Shift to right.
inline Integer& Shift2Right( const unsigned __int64 uiTimes )
{
   if( !IsZero() && (0 != uiTimes) )
   {
      const unsigned __int64 uiDelSegments( uiTimes / 64 );

      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
      // Remove one segment from the head, i.e. least segnificant part, for every 64 bit shift to the right.
      for( unsigned __int64 uiSegment = 0; (uiSegment < uiDelSegments) && !IsEmpty(); uiSegment++ )
      {
         delete RemoveHead();
      }

      if( !IsEmpty() )
      {
         const BYTE b1BitIncShift( uiTimes % 64 );

         if( 0 != b1BitIncShift )
         {
            // Division by 64 cannot have remiander bigger than 63.
            MASSERT( b1BitIncShift < 64 );

            // Head - LSB, Tail - MSB.
            unsigned __int64 uiCarry( 0 );
```

```
               for( DWORD dwIndex = GetCount() - 1; (DWORD)-1 != dwIndex; dwIndex-- )
               {
                  const unsigned __int64 bNewCarry( *(*this)[dwIndex] << (64 - b1BitIncShift) );

                  *(*this)[dwIndex] >>= b1BitIncShift;
                  *(*this)[dwIndex] |=  uiCarry;

                  uiCarry = bNewCarry;
               }
            }
         }
      }

      // Lose any newly created leading zeros or emptied object.
      Normalize();

      return( *this );
   }


   // This method attempts to represent the integer number as double precision floating point number.
   // Note that a number from the Integer class may be much larger than the larger number that a double
   // precision floating point could represent. Also, note that floating point numbers are not evenly
   // distributed having irregularly sized gaps between them. In difference the Integer numbers are
   // evenly distributed. This means that Integer numbers could be in fact very rarely represented
   // exactly using a floating point number. This function is useful for displaying Integer numbers.
   double GetAsDouble() const
   {
      // The algorithm is to multiply the double 1 by *this and keep the result in a double variable.
      if( IsZero() )
      {
         return( 0 );
      }


      Integer  iFactor( *this );
      double   dThis( 0 );
      double   dFactor( 1 );

      while( 0 != iFactor )
      {
         if( iFactor.Shift2Right() )
         {
            // There was a right carry.
            dThis += dFactor;
         }

         dFactor *= 2;
      }

      return( dThis );
   }


private:
   // Shift to right with one bit. Returns true if one has been
   // lost by the shift and false if zero was removed (lost).
   inline bool Shift2Right()
   {
      bool bRightCarry( false );

      // Head - LSB, Tail - MSB.
      for( DWORD dwIndex = GetCount() - 1; (DWORD)-1 != dwIndex; dwIndex-- )
      {
         const bool bNewRightCarry( 1 & (*(*this)[dwIndex]) );

         (*(*this)[dwIndex]) >>= 1;

         if( bRightCarry )
         {
            (*(*this)[dwIndex]) |= (((unsigned __int64)1) << 63);
         }

         bRightCarry = bNewRightCarry;
      }

      // Lose any newly created leading zeros.
      Normalize();

      return( bRightCarry );
   }
```

```
inline void AddDecimal( MList< BYTE >& listResult, const MList< BYTE >& list2Power ) const
{
   const DWORD dwLoopCount( MMAX< DWORD, DWORD, DWORD >( listResult.GetCount(), list2Power.GetCount() ) );

   BYTE b1Carry( 0 );
   for( DWORD dwSegment = 0; dwSegment < dwLoopCount; dwSegment++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Head-to-Tail.
      BYTE b1This( b1Carry );
      if( listResult.GetCount() > dwSegment )
      {
         b1This += *listResult[dwSegment];
      }

      if( list2Power.GetCount() > dwSegment )
      {
         b1This += *list2Power[dwSegment];
      }


      if( 9 < b1This )
      {
         b1This -= 10;
         b1Carry = 1;
      }
      else
      {
         b1Carry = 0;
      }


      if( listResult.GetCount() > dwSegment )
      {
         *listResult[dwSegment] = b1This;
      }
      else
      {
         listResult.AddTail( new BYTE( b1This ) );
      }
   }

   if( 0 != b1Carry )
   {
      listResult.AddTail( new BYTE( b1Carry ) );
   }
}


inline void MultiplyDecimalBy2( MList< BYTE >& list2Power ) const
{
   BYTE b1Carry( 0 );

   // Head is LSS, Tail is MSS
   for( DWORD dwSegment = 0; dwSegment < list2Power.GetCount(); dwSegment++ )
   {
      BYTE b1New = 2 * (*list2Power[dwSegment]) + b1Carry;

      if( 9 < b1New )
      {
         *list2Power[dwSegment] = b1New - 10;

         b1Carry = 1;
      }
      else
      {
         *list2Power[dwSegment] = b1New;

         b1Carry = 0;
      }
   }

   if( 0 != b1Carry )
   {
      list2Power.AddTail( new BYTE( b1Carry ) );
   }
}


inline void IncrementFromSegment( const DWORD dwSegment )
```

```
{
   if( dwSegment < GetCount() )
   {
      // Head LSB, Tail MSB.
      if( (unsigned __int64)-1 > *(*this)[dwSegment] )
      {
         (*(*this)[dwSegment])++;
      }
      else
      {
         (*(*this)[dwSegment]) = 0;

         IncrementFromSegment( dwSegment + 1 );
      }
   }
   else
   {
      AddTail( new unsigned __int64( 1 ) );
   }
}


// External operators - friend operators.


// Standard comparison operator equal to.
friend bool operator ==( const Integer& iLeft, const Integer& iRight )
{
   if( iLeft.GetCount() != iRight.GetCount() )
   {
      return( false );
   }

   for( DWORD dwIndex = 0; dwIndex < iLeft.GetCount(); dwIndex++ )
   {
      if( *iLeft[dwIndex] != *iRight[dwIndex] )
      {
         return( false );
      }
   }

   return( true );
}


// Standard comparison operator not equal to.
friend bool operator !=( const Integer& rLeft, const Integer& rRight )
{
   return( !::operator==( rLeft, rRight ) );
}


// Standard comparison operator less than.
friend inline bool operator <( const Integer& iLeft, const Integer& iRight )
{
   if( iLeft.GetCount() == iRight.GetCount() )
   {
      // When equal size-in-part compare part by part. Compare from Tail to Head, since
      // the Head is Least Significant Part, and the Tail is Most Significant Part.
      for( DWORD dwIndex = iLeft.GetCount() - 1; (DWORD)-1 > dwIndex; dwIndex-- )
      {
         if( *iLeft[dwIndex] != *iRight[dwIndex] )
         {
            return( *iLeft[dwIndex] < *iRight[dwIndex] );
         }
      }

      // All parts are equal, so false.
      return( false );
   }

   return( iLeft.GetCount() < iRight.GetCount() );
}


// Standard comparison operator greater than.
friend inline bool operator >( const Integer& iLeft, const Integer& iRight )
{
   if( iLeft.GetCount() == iRight.GetCount() )
   {
      // When equal size-in-part compare part by part. Compare from Tail to Head, since
      // the Head is Least Significant Part, and the Tail is Most Significant Part.
      for( DWORD dwIndex = iLeft.GetCount() - 1; (DWORD)-1 > dwIndex; dwIndex-- )
      {
         if( *iLeft[dwIndex] != *iRight[dwIndex] )
         {
            return( *iLeft[dwIndex] > *iRight[dwIndex] );
         }
      }

      // All parts are equal, so false.
      return( false );
   }

   return( iLeft.GetCount() > iRight.GetCount() );
}


// Standard comparison operator less than or equal to.
friend inline bool operator <=( const Integer& iLeft, const Integer& iRight )
{
   if( iLeft.GetCount() == iRight.GetCount() )
   {
      // When equal size-in-part compare part by part. Compare from Tail to Head, since
      // the Head is Least Significant Part, and the Tail is Most Significant Part.
      for( DWORD dwIndex = iLeft.GetCount() - 1; (DWORD)-1 > dwIndex; dwIndex-- )
      {
         if( *iLeft[dwIndex] != *iRight[dwIndex] )
         {
            return( *iLeft[dwIndex] <= *iRight[dwIndex] );
         }
      }

      // All parts are equal, so equal.
      return( true );
   }

   return( iLeft.GetCount() < iRight.GetCount() );
}


// Standard comparison operator greater than or equal to.
friend inline bool operator >=( const Integer& iLeft, const Integer& iRight )
{
   if( iLeft.GetCount() == iRight.GetCount() )
   {
      // When equal size-in-part compare part by part. Compare from Tail to Head, since
      // the Head is Least Significant Part, and the Tail is Most Significant Part.
      for( DWORD dwIndex = iLeft.GetCount() - 1; (DWORD)-1 > dwIndex; dwIndex-- )
      {
         if( *iLeft[dwIndex] != *iRight[dwIndex] )
         {
            return( *iLeft[dwIndex] >= *iRight[dwIndex] );
         }
      }

      // All parts are equal, so equal.
      return( true );
   }

   return( iLeft.GetCount() > iRight.GetCount() );
}


// Standard arithmetic operator plus.
friend inline Integer operator +( const Integer& iSummandL, const Integer& iSummandR )
{
   // Create an empty result.
   Integer iResult( iSummandL, 0 );

   const DWORD dwLoopCount( MMAX< DWORD, DWORD, DWORD >( iSummandL.GetCount(), iSummandR.GetCount() ) );

   unsigned __int64 uiCarry( 0 );
   for( DWORD dwIndex = 0; dwIndex < dwLoopCount; dwIndex++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.

      unsigned __int64 uiLeft( 0 );
      if( iSummandL.GetCount() > dwIndex )
      {
         uiLeft = *iSummandL[dwIndex];
      }
      unsigned __int64 uiRight( 0 );
      if( iSummandR.GetCount() > dwIndex )
      {
         uiRight = *iSummandR[dwIndex];
      }

      iResult.AddTail( new unsigned __int64( uiLeft + uiRight + uiCarry ) );

      // Find out if there is a carry.
      unsigned __int64 uiOverflow( (unsigned __int64)-1 - uiLeft );

      if( uiOverflow < uiRight )
      {
         // There was a carry;
         uiCarry = 1;

         continue;
      }

      uiOverflow -= uiRight;

      uiCarry = uiOverflow < uiCarry ? 1 : 0;
   }

   if( 0 != uiCarry )
   {
      MASSERT( 1 == uiCarry );

      iResult.AddTail( new unsigned __int64( uiCarry ) );
   }

   if( 0 == iResult.GetCount() )
   {
      iResult.AddTail( new unsigned __int64( 0 ) );
   }

   return( iResult );
}


// Standard arithmetic operator minus. When subtracting larger number from smaller
// the result wraps up and has the size of the larger number, i.e. A > B; C = B - A;
// [size C == size A] = [size B] - [size A], C = 0 - ( A - B ).
friend inline Integer operator -( Integer iMinuend, Integer iSubtrahend )
{
   // Create an empty result.
   Integer iResult( iMinuend, 0 );

   const DWORD dwLoopCount( MMAX< DWORD, DWORD, DWORD >( iMinuend.GetCount(), iSubtrahend.GetCount() ) );

   // Create equal elements countinteger numbers.
   iMinuend = Integer( iMinuend, dwLoopCount );
   iSubtrahend = -Integer( iSubtrahend, dwLoopCount );

   unsigned __int64 uiCarry( 0 );
   for( DWORD dwIndex = 0; dwIndex < dwLoopCount; dwIndex++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
      const unsigned __int64 uiLeft( *iMinuend[dwIndex] );
      const unsigned __int64 uiRight( *iSubtrahend[dwIndex] );

      iResult.AddTail( new unsigned __int64( uiLeft + uiRight + uiCarry ) );

      // Find out if there is a carry.
      unsigned __int64 uiOverflow( (unsigned __int64)-1 - uiLeft );

      if( uiOverflow < uiRight )
      {
         // There was a carry;
         uiCarry = 1;

         continue;
      }

      uiOverflow -= uiRight;

      uiCarry = uiOverflow < uiCarry ? 1 : 0;
   }

   if( 0 == iResult.GetCount() )
```

```
      {
         iResult.AddTail( new unsigned __int64( 0 ) );
      }

      // Note that in subtraction I ignore the most significant Carry.
      return( iResult.Normalize() );
   }


   // Standard arithmetic operator multiplication.
   friend inline Integer operator *( const Integer& iFactorL, const Integer& iFactorR )
   {
      // Fast multiplication algorithm. Push the shorther to the right amending the longer.
      Integer iShorter( iFactorL < iFactorR ? iFactorL : iFactorR );
      Integer iLonger( iFactorL > iFactorR ? iFactorL : iFactorR );

      // Create an empty result.
      Integer iResult( iFactorL, 0 );

      while( 0 != iShorter )
      {
         if( iShorter.Shift2Right() )
         {
            // There was a right carry.
            iResult += iLonger;
         }

         iLonger.Shift2Left( 1 );
      }

      if( 0 == iResult.GetCount() )
      {
         iResult.AddTail( new unsigned __int64( 0 ) );
      }

      return( iResult );
   }


   // Standard arithmetic operator division. The division operator returns a pair of specialized integers
   // for quotient and reminder. This operator throws NumberException exception when dividing by zero.
   friend inline Pair< specialize< Integer as sX::Quotient >, specialize< Integer as sX::Reminder > > operator /( Integer iDividend, Integer
iDivisor )
   {
      NumberException< Integer, true >::TestDivisionByZero( iDivisor );

      // Fast division algorithm. Push the shorther to the right amending the longer.
      unsigned __int64 uiMS1_Divisor( iDivisor.GetMostSignificantOneIndex() );
      specialize< Integer as sX::Quotient > iQuotient( specialize< Integer as sX::Quotient >::Initialize( 0 ) );

      while( iDividend >= iDivisor )
      {
         unsigned __int64 uiMS1_Divident( iDividend.GetMostSignificantOneIndex() );
         unsigned __int64 uiOrderDifference( uiMS1_Divident - uiMS1_Divisor );

         Integer iExpandedDivisor( iDivisor );
         iExpandedDivisor.Shift2Left( uiOrderDifference );

         if( iDividend < iExpandedDivisor )
         {
            uiOrderDifference--;
            iExpandedDivisor.Shift2Right();
         }

         iQuotient += Integer( 1 ).Shift2Left( uiOrderDifference );
         iDividend -= iExpandedDivisor;
      }

      return( Pair< specialize< Integer as sX::Quotient >, specialize< Integer as sX::Reminder > >( iQuotient, specialize< Integer as
sX::Reminder >::Initialize( iDividend ) ) );
   }


   // Standard arithmetic operator modulo.
   friend inline Integer operator %( const Integer& iDividend, const Integer& iDivisor )
   {
      return( operator /( iDividend, iDivisor ).GetValueRef() );
   }


   // Bitwise AND operator.
```

```
friend inline Integer operator &( const Integer& iGateL, const Integer& iGateR )
{
   // Create an empty result.
   Integer iResult( iGateL, 0 );

   const DWORD dwLoopCount( MMIN< DWORD, DWORD, DWORD >( iGateL.GetCount(), iGateR.GetCount() ) );

   for( DWORD dwIndex = 0; dwIndex < dwLoopCount; dwIndex++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
      iResult.AddTail( new unsigned __int64( *iGateL[dwIndex] & *iGateR[dwIndex] ) );
   }

   MASSERT( 0 != iResult.GetCount() );

   return( iResult );
}


// Bitwise OR operator.
friend inline Integer operator |( const Integer& iGateL, const Integer& iGateR )
{
   // Create an empty result.
   Integer iResult( iGateL, 0 );

   const DWORD dwLoopCount( MMAX< DWORD, DWORD, DWORD >( iGateL.GetCount(), iGateR.GetCount() ) );

   for( DWORD dwIndex = 0; dwIndex < dwLoopCount; dwIndex++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.

      unsigned __int64 uiLeft( 0 );
      if( iGateL.GetCount() > dwIndex )
      {
         uiLeft = *iGateL[dwIndex];
      }

      unsigned __int64 uiRight( 0 );
      if( iGateR.GetCount() > dwIndex )
      {
         uiRight = *iGateR[dwIndex];
      }

      iResult.AddTail( new unsigned __int64( uiLeft | uiRight ) );
   }

   return( iResult );
}


// Bitwise XOR operator.
friend inline Integer operator ^( const Integer& iGateL, const Integer& iGateR )
{
   // Create an empty result.
   Integer iResult( iGateL, 0 );

   const DWORD dwLoopCount( MMAX< DWORD, DWORD, DWORD >( iGateL.GetCount(), iGateR.GetCount() ) );

   for( DWORD dwIndex = 0; dwIndex < dwLoopCount; dwIndex++ )
   {
      // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.

      unsigned __int64 uiLeft( 0 );
      if( iGateL.GetCount() > dwIndex )
      {
         uiLeft = *iGateL[dwIndex];
      }

      unsigned __int64 uiRight( 0 );
      if( iGateR.GetCount() > dwIndex )
      {
         uiRight = *iGateR[dwIndex];
      }

      iResult.AddTail( new unsigned __int64( uiLeft ^ uiRight ) );
   }


   // XOR may cause non zero operands to produce zero result
   iResult.Normalize();
```

```
      return( iResult );
   }


   // Shift to left operator. This operator may allocate significant amount of system resources if used without care. Passing
   // large number of left shifts will incur adding a new 64 bit segment to the result for every 64 units in the shift request.
   friend inline Integer operator <<( const Integer& iThisObject, const Integer& iShiftCount )
   {
      if( iThisObject.IsZero() )
      {
         return( Integer( 0 ) );
      }

      Integer iResult( iThisObject );

      if( !iShiftCount.IsZero() )
      {
         const Pair< specialize< Integer as sX::Quotient >, specialize< Integer as sX::Reminder > > prDivision( iShiftCount / 64 );

         // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
         // Add one empty segment at the head, i.e. least significant part, for every 64 bit shift to the left.
         for( Integer iSegment = 0; iSegment < prDivision.GetLabelRef(); iSegment++ )
         {
            iResult.AddHead( new unsigned __int64( 0 ) );
         }

         // Division by 64 cannot have remiander bigger than 63.
         MASSERT( prDivision.GetValueRef() < 64 );

         // Shift for the remainder, 64 is well held in one segment.
         iResult.Shift2Left( *prDivision.GetValueRef().GetTail() );
      }

      return( iResult );
   }


   // Shift to right operator.
   friend inline Integer operator >>( const Integer& iThisObject, const Integer& iShiftCount )
   {
      if( iThisObject.IsZero() )
      {
         return( Integer( 0 ) );
      }

      Integer iResult( iThisObject );

      if( !iShiftCount.IsZero() )
      {
         const Pair< specialize< Integer as sX::Quotient >, specialize< Integer as sX::Reminder > > prDivision( iShiftCount / 64 );

         // The Head is Least Significant Part, and the Tail is Most Significant Part so compute Headt-to-Tail.
         // Remove one segment from the head, i.e. least significant part, for every 64 bit shift to the right.
         for( Integer iSegment = 0; (iSegment < prDivision.GetLabelRef()) && (0 < iResult.GetCount()); iSegment++ )
         {
            delete iResult.RemoveHead();
         }

         // Division by 64 cannot have remiander bigger than 63.
         MASSERT( prDivision.GetValueRef() < 64 );

         // Shift for the remainder, 64 is well held in one segment.
         iResult.Shift2Right( *prDivision.GetValueRef().GetTail() );
      }

      return( iResult );
   }
};
```

4.2.2. The Rational Class Template

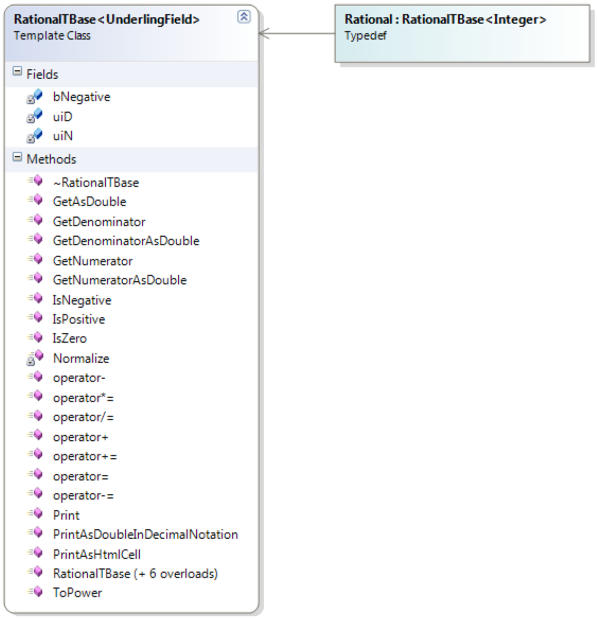
The RationalTBase template class represents rational numbers over an unspecified specializer which must be at least a commutative ring with identity (CRI). The rational entities are represented by a sign and an ordered pair of numerator and denominator. The objects are always normalized, i.e. the numerator and denominator are always coprime. When RationalTBase is specialized with the Integer class (which represents $\mathbb{N}$ (up to available memory)) the resulting type represents $\mathbb{Q}$ (up to available memory), i.e. the objects of type RationalTBase< Integer > are rational numbers with arbitrary precision and even distribution, thus they are perfect for precise computations. It is possible to use other types of underlying CRIs, such as unsigned char - $\mathbb{Z}_{2^8}$, unsigned short - $\mathbb{Z}_{2^{16}}$, unsigned int - $\mathbb{Z}_{2^{32}}$, unsigned long - $\mathbb{Z}_{2^{32}}$, unsigned __int64 - $\mathbb{Z}_{2^{64}}$, etc. or applicable user defined types. Being a commutative ring with identity the specializer is expected to have all appropriate operators defined.

The behaviour of the type is determined by its own nature and also by the behaviour of the underlying CRI. For example overflows (when applicable) may or may not be signalled depending on the behaviour of the underlying ring. RationalTBase throws NumberException only when dividing by zero. This behaviour may change in future removing all exceptions altogether although this is not likely, or new operator with alternative behaviour will be provided.

RationalTBase has Boolean and bitwise operators NOT overloaded purposefully. The reason is that rational numbers are not Boolean or bit-field entities and therefore should not implicitly represent them. Objects from the rational class are able to print themselves in various ways.

The Rational is a type definition, specialization of RationalTBase with the Integer class representing arbitrary precision rational numbers. It is ideal for use when precise computations are required. For example the usually used for computation floating point numbers have overflows and underflows and rarely represent a number precisely. They have accumulative errors, thus the more they are used the bigger the error. In addition they are denser around the zero and further apart from each other as going away from the zero. The Rational have no such flaws. Another advantage is that the Rational is itself a commutative ring with identity (CRI) with all appropriate operators and thus can be used whenever ring is needed.

Figure 10. Class diagram of the RationalTBBase template and Rational specialization type from the Proper Numbers Library.



Listing 7. The RationalTBase template and Rational specialization type from the Proper Numbers Library.

```
//
//
// Rational.h: Rational with arbitrary precision.
//
//                Proper Numbers Library
//
// © Copyright 2011 - 2011 by Miroslav Bonchev Bonchev. All rights reserved.
//
//
//******************************************************************************************************


// Open Source License – The MIT License
//
//
// {your product} uses the: Proper Numbers Library – Copyright © 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
// {your product} uses the: Object Specialization Model – Copyright © 2009 - 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
// associated  documentation files  (the "Software"),  to deal  in the Software without restriction,
// including  without  limitation the rights  to use,  copy,  modify,  merge,  publish,  distribute,
// sublicense,  and/or sell copies of the Software,  and to permit persons to  whom the  Software is
// furnished to do so, subject to the following conditions:
//
// The  above  copyright  notice  and  this  permission  notice  shall be  included in all copies or
// substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT
// NOT  LIMITED  TO  THE  WARRANTIES  OF  MERCHANTABILITY,  FITNESS  FOR  A  PARTICULAR  PURPOSE AND
// NONINFRINGEMENT.  IN NO EVENT SHALL  THE  AUTHORS  OR  COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
// DAMAGES OR OTHER LIABILITY,  WHETHER IN AN ACTION OF CONTRACT,  TORT OR OTHERWISE,  ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.


//__________________________________________________________________________________________________
// This software is OSI Certified Open Source Software.
// OSI Certified is a certification mark of the Open Source Initiative.


#pragma once


#include "stdafx.h"
#include "Common.h"
#include "StringEx.h"
#include "Specialize.h"
#include "Integer.h"


#define Minus_Char_TextSize         L"22"
#define Rational_Notation_TextSize  L"12"
#define Decimal_Notation_TextSize   L"12"


// The RationalTBase template class represents signed rational numbers over an unspecified specializer filed.
// The Rational is a specialization of RationalTBase with the Integer class representing arbitrary precision
// signed rational numbers.
//
// Objects from type Rational expand arbitrarily large up to available memory and are able to accommodate
// arbitrarily large numbers with arbitrary precision (up to the available memory).
//
// Some points of interest:
//
// The rational numbers are represented by a sign and an ordered pair of numerator and denominator elements
// of the underlying commutative ring with identity CRI (the specialiser of the template). When the template
// is specialized with the Integer class, which represents an arbitrarily large unsigned integer numbers, the
// Rational numbers represented by this class become with arbitrary precision and even distribution numbers,
// thus become perfect for precise computations. It is possible to use other types of underlying rings (CRI),
// such as BYTE, thus creating Rational Numbers over Z[2^8], unsigned short achieving rational numbers over
// Z[2^16], unsigned int achieving rational numbers over Z[2^32], unsigned __int64 achieving rational numbers
// over Z[2^64], etc.
//
// Overflow may or may not be signalled depending on the behaviour of the underlying CRI. For example since
// the numbers from the Integer class never overflow, but instead they grow thus maintaining their integrity,
// Rational specialized with Integer never overflows and is always correct. The build in types on the other
// hand ignore any overflows, hence Rational specialized with build in type may become erroneous without
// signalling the overflow. Other specializer types may or may not signal overflow in accordance with their
// behaviour. Numbers of Rational type are always normalized so that the numerator and denominator are always
// coprime.
//
// Objects from this type throw NumberException only when dividing by zero. This behaviour
// may change in future removing all exceptions altogether although this is not likely, or new operator
// with alternative behaviour will be provided.
//
// The logical operators are NOT overloaded purposefully. The reason is that rational numbers are NOT Boolean
// entities and therefore they should not implicitly represent logical values. Should one want to use them
// as Boolean variables they can use the IsZero() method and the comparison operators appropriately.
//
// Bitwise operators are NOT overloaded purposefully as not applicable for a rational type.
//
// Objects from the rational class are able to print themselves in various ways.
template< class UnderlingCRI >
class RationalTBase
{
private:
   bool bNegative;
   specialize< UnderlingCRI as sX::Numerator >   uiN;
   specialize< UnderlingCRI as sX::Denominator > uiD;


public:
   // Default constructor - initializes the object with zero.
   RationalTBase()
      :  bNegative( false ),
         uiN( specialize< UnderlingCRI as sX::Numerator >::Initialize( 0 ) ),
         uiD( specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 ) )
   {
   }


   // Constructor from signed integer.
   RationalTBase( const __int64 nInteger )
      :  bNegative( 0 > nInteger ),
         uiN( specialize< UnderlingCRI as sX::Numerator >::Initialize( MABSsat< UnderlingCRI, __int64 >( nInteger ) ) ),
         uiD( specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 ) )
   {
      Normalize();
   }


   // Copy constructor.
   RationalTBase( const RationalTBase< UnderlingCRI >& rational )
      :  bNegative( rational.bNegative ),
         uiN( rational.uiN ),
         uiD( rational.uiD )
   {
   }


   // Constructor from unsigned Integer.
   RationalTBase( const specialize< UnderlingCRI as sX::Numerator >& uiNumerator )
      :  bNegative( false ),
         uiN( uiNumerator ),
         uiD( specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 ) )
   {
      Normalize();
   }


   // Constructor from signed and numerator, and denominator.
   RationalTBase( const bool bNegative,
                  const specialize< UnderlingCRI as sX::Numerator >& uiNumerator )
      :  bNegative( bNegative ),
         uiN( uiNumerator ),
         uiD( specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 ) )
   {
      Normalize();
   }


   // Constructor from unsigned numerator and denominator.
   RationalTBase( const specialize< UnderlingCRI as sX::Numerator >& uiNumerator,
                  const specialize< UnderlingCRI as sX::Denominator >& uiDenominator )
      :  bNegative( false ),
         uiN( uiNumerator ),
         uiD( uiDenominator )
   {
      NumberException< UnderlingCRI, true >::TestDivisionByZero( uiD );

      Normalize();
   }


   // Constructor from signed and numerator, and denominator.
   RationalTBase( const bool bNegative,
                  const specialize< UnderlingCRI as sX::Numerator >& uiNumerator,
                  const specialize< UnderlingCRI as sX::Denominator >& uiDenominator )
      :  bNegative( bNegative ),
         uiN( uiNumerator ),
         uiD( uiDenominator )
   {
      NumberException< UnderlingCRI, true >::TestDivisionByZero( uiD );
      Normalize();
   }


   // Destructor.
   ~RationalTBase()
   {
   }


   // Standard operator equal.
   RationalTBase< UnderlingCRI >& operator=( const RationalTBase< UnderlingCRI >& rational )
   {
      bNegative = rational.bNegative;
      uiN = rational.uiN;
      uiD = rational.uiD;

      return( *this );
   }


   // Standard operator +=
   RationalTBase< UnderlingCRI > operator +=( const RationalTBase< UnderlingCRI >& rSummand )
   {
      return( *this = *this + rSummand );
   }


   // Standard operator -=
   RationalTBase< UnderlingCRI > operator -=( const RationalTBase< UnderlingCRI >& rSubtrahend )
   {
      return( *this = *this - rSubtrahend );
   }


   // Standard operator *=
   RationalTBase< UnderlingCRI > operator *=( const RationalTBase< UnderlingCRI >& rFactor )
   {
      return( *this = *this * rFactor );
   }


   // Standard operator /=
   RationalTBase< UnderlingCRI > operator /=( const RationalTBase< UnderlingCRI >& rDivisor )
   {
      return( *this = *this / rDivisor );
   }


   // Returns an rational power of this integer.
   inline RationalTBase< UnderlingCRI > ToPower( const Integer& uiPower ) const
   {
      RationalTBase< UnderlingCRI > rThis( 1 );

      for( Integer uiCount = 0; uiCount < uiPower; uiCount += 1 )
      {
         rThis = rThis * *this;
      }

      return( rThis );
   }


   // Unary plus (returns the operand itself).
   RationalTBase< UnderlingCRI > operator+() const
   {
      return( *this );
   }


   // Unary operator negation - inverts the signd.
   RationalTBase< UnderlingCRI > operator-() const
   {
      return( IsZero() ? *this : RationalTBase< UnderlingCRI >( !bNegative, uiN, uiD ) );
   }


   // Test if the number is zero.
   bool IsZero() const
   {
      return( 0 == uiN );
   }


   // Test if the number is positive.
   bool IsPositive() const
   {
      return( !bNegative );
   }


   // Test if the number is negative.
   bool IsNegative() const
   {
      return( bNegative );
   }


   // Returns the numerator.
   const UnderlingCRI& GetNumerator() const
   {
      return( uiN );
   }


   // Returns the denominator.
   const UnderlingCRI& GetDenominator() const
   {
      return( uiD );
   }


   // This method returns the numerator as double precision floating point number. This method is
   // intended for use when the Rational is specialized with the Integer class. If used with another
   // specializer type, it must have method double GetAsDouble() const. Note that a number from the
   // Integer class may be much larger than the larger number that a double precision floating point
   // could represent. Also, note that floating point numbers are not evenly distributed having
   // irregularly sized gaps between them. In difference the Integer numbers are evenly distributed.
   // This means that Integer numbers could be in fact very rarely represented exactly using a
   // floating point number. This function is useful for displaying Integer numbers.
   double GetNumeratorAsDouble() const
   {
      return( uiN.GetAsDouble() );
   }


   // This method returns the denominator as double precision floating point number. This method is
   // intended for use when the Rational is specialized with the Integer class. If used with another
   // specializer type, it must have method double GetAsDouble() const. Note that a number from the
   // Integer class may be much larger than the larger number that a double precision floating point
   // could represent. Also, note that floating point numbers are not evenly distributed having
   // irregularly sized gaps between them. In difference the Integer numbers are evenly distributed.
   // This means that Integer numbers could be in fact very rarely represented exactly using a
   // floating point number. This function is useful for displaying Integer numbers.
   double GetDenominatorAsDouble() const
   {
      return( uiD.GetAsDouble() );
   }


   // The method returns the rational number as a double precision floating point number. This method
   // is intended for use when the Rational is specialized with the Integer class. If used with another
   // specializer type, it must have method double GetAsDouble() const. Note that a number from the
   // Rational class may be much larger than the larger number that a double precision floating point
   // could represent and is also PRECISE. Also, note that floating point numbers are not evenly
   // distributed having irregularly sized gaps between them. In difference the Rational specialized
   // with Integer are evenly distributed and PRECISE. This means that Rational specialized with Integer
   // number could be in fact very rarely represented using a floating point number. This function is
   // useful for displaying Rational numbers for informative purpoces.
   double GetAsDouble() const
   {
      return( (bNegative ? -1 : 1) * GetNumeratorAsDouble() / GetDenominatorAsDouble() );
   }


   // This method returns a signed pair +/-( numerator, denominator) representing the rational number exactly.
   // This method is intended for use when the Rational is specialized with the Integer class. If used with
   // another specializer type, it must have method string Print() const, where string must have LPCTSTR
   // overloaded operator. Although this method returns the exact rational number, in some circumstances
   // one may want to also consider using the float representations for informative purposes as their results
   // may be easier to interpret.
   MStringEx< TCHAR > Print() const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%c(%s,%s)"), bNegative ? TCHAR('-') : TCHAR('+'),
(LPCTSTR)uiN.Print(), (LPCTSTR)uiD.Print() ) );
   }


   // This method returns the double representation of the rational number as a string in decimal notation.
   // The parameter sets the required number of digits after the decimal point, max number is 16. Note that
   // the floating point representation is most likely inaccurate to some degree depending on the represented
   // number. Floating point numbers cannot represent proper rational numbers satisfactory.
   MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%+.") + MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%d"),
(0x0F & (b1Precision-1))+1 ) + TEXT("f"), GetAsDouble() ) );
```

```
   }


   // This method returns a HTML table containing the rational number in rational notation and floating point
   // notation. This method is intended for use when the Rational is specialized with the Integer class. If
   // used with another specializer type, it must have defined the methods used by this function. The parameter
   // passed to the method is string which is inserted in the HTML table. Thus one can pass additional
   // information to be printed together with the rational number, inside the returned HTML table.
   MStringEx< TCHAR > PrintAsHtmlCell() const
   {
      // |-table 1---------------------------------------|-----------------------------------------------|
      // | |-table 2-----------------------------------| |                                               |
      // | |     sign  |  table 3 - rational notation  | |             double approximation              |
      // | |-------------------------------------------| |                                               |
      // |-----------------------------------------------|-----------------------------------------------|
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF,
                                  L"<table border='0' cellpadding='0' cellspacing='0' style='text-align: right;'>\
                                      <tr><td style='padding-right:10px; text-align: right; width: 500px;'>\
                                        <table border='0' cellpadding='0' cellspacing='0'>\
                                          <tr>\
                                            <td style='font-size:" Minus_Char_TextSize L"px;'>%c&nbsp</td>\
                                            <td><table border='0' cellpadding='0' cellspacing='0' style='text-align: right; font-
size:" Rational_Notation_TextSize L"px;'><tr><td style='border-bottom:solid black
thin;'>%s</td></tr><tr><td>%s</td></tr></table></td>\
                                          </tr>\
                                        </table>\
                                        </td>\
                                        <td style='padding-left:20px; text-align: left; font-size:" Decimal_Notation_TextSize L"px;
width: 200px;'>%s</td>\
                                      </tr></table>", bNegative ? TCHAR('-') : TCHAR('+'), (LPCTSTR)uiN.Print(),
(LPCTSTR)uiD.Print(), (LPCTSTR)PrintAsDoubleInDecimalNotation( 16 ) ) );
   }


private:
   void Normalize()
   {
      if( 0 != uiN )
      {
         UnderlingCRI uiDivident( MMAX< UnderlingCRI, UnderlingCRI >( uiN, uiD ) );
         UnderlingCRI uiDivisor(  MMIN< UnderlingCRI, UnderlingCRI >( uiN, uiD ) );

         UnderlingCRI uiQuotient( uiDivident.DivideBy_ReturnQuotient( uiDivisor ) );

         for( UnderlingCRI uiReminder( uiDivident - uiDivisor * uiQuotient ); 0 != uiReminder; uiReminder = uiDivident - uiDivisor *
uiQuotient )
         {
            uiDivident = uiDivisor;
            uiDivisor  = uiReminder;
            uiQuotient = uiDivident.DivideBy_ReturnQuotient( uiDivisor );
         }

         if( 1 != uiDivisor )
         {
            uiN /= uiDivisor;
            uiD /= uiDivisor;
         }
      }
      else
      {
         bNegative = false;
         uiD = specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 );
      }
   }


   // External operators - friend operators.


   // Standard comparison operator equal to.
   friend inline bool operator ==( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      return( (rLeft.bNegative == rRight.bNegative) && (rLeft.uiN == rRight.uiN) && (rLeft.uiD == rRight.uiD) );
   }


   // Standard comparison operator not equal to.
   friend inline bool operator !=( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      return( (rLeft.bNegative != rRight.bNegative) || (rLeft.uiN != rRight.uiN) || (rLeft.uiD != rRight.uiD) );
   }


   // Standard comparison operator less than.
   friend inline bool operator <( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      if( rLeft.bNegative ^ rRight.bNegative )
      {
         // One is negative, the other positive.
         return( rLeft.bNegative );
      }

      return( rLeft.bNegative ? (rLeft.uiN * rRight.uiD) > (rRight.uiN * rLeft.uiD) : (rLeft.uiN * rRight.uiD) < (rRight.uiN *
rLeft.uiD) );
   }


   // Standard comparison operator greater than.
   friend inline bool operator >( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      if( rLeft.bNegative ^ rRight.bNegative )
      {
         // One is negative, the other positive.
         return( rRight.bNegative );
      }

      return( rLeft.bNegative ? (rLeft.uiN * rRight.uiD) < (rRight.uiN * rLeft.uiD) : (rLeft.uiN * rRight.uiD) > (rRight.uiN *
rLeft.uiD) );
   }


   // Standard comparison operator less than or equal to.
   friend inline bool operator <=( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      if( rLeft.bNegative ^ rRight.bNegative )
      {
         // One is negative, the other positive.
         return( rLeft.bNegative );
      }

      return( rLeft.bNegative ? (rLeft.uiN * rRight.uiD) >= (rRight.uiN * rLeft.uiD) : (rLeft.uiN * rRight.uiD) <= (rRight.uiN *
rLeft.uiD) );
   }


   // Standard comparison operator greater than or equal to.
   friend inline bool operator >=( const RationalTBase< UnderlingCRI >& rLeft, const RationalTBase< UnderlingCRI >& rRight )
   {
      if( rLeft.bNegative ^ rRight.bNegative )
      {
         // One is negative, the other positive.
         return( rRight.bNegative );
      }

      return( rLeft.bNegative ? (rLeft.uiN * rRight.uiD) <= (rRight.uiN * rLeft.uiD) : (rLeft.uiN * rRight.uiD) >= (rRight.uiN *
rLeft.uiD) );
   }


   // Standard arithmetic operator plus.
   friend inline RationalTBase< UnderlingCRI > operator +( const RationalTBase< UnderlingCRI >& iSummandL, const RationalTBase<
UnderlingCRI >& iSummandR )
   {
      const UnderlingCRI uiL( iSummandL.uiN * iSummandR.uiD );
      const UnderlingCRI uiR( iSummandR.uiN * iSummandL.uiD );

      Rational r;

      if( iSummandL.bNegative ^ iSummandR.bNegative )
      {
         // Both operands are have different signs.
         r.uiN = specialize< UnderlingCRI as sX::Numerator >::Initialize( MMAX< UnderlingCRI, UnderlingCRI, UnderlingCRI >( uiL, uiR
) - MMIN< UnderlingCRI, UnderlingCRI, UnderlingCRI >( uiL, uiR ) );
         r.bNegative = uiL > uiR ? iSummandL.bNegative : iSummandR.bNegative;
      }
      else
      {
         // Both operands are the same sign.
         r.uiN = specialize< UnderlingCRI as sX::Numerator >::Initialize( uiL + uiR );
         r.bNegative = iSummandL.bNegative;
      }

      r.uiD = specialize< UnderlingCRI as sX::Denominator >::Initialize( iSummandL.uiD * iSummandR.uiD );

      r.Normalize();

      return( r );
   }


   // Standard arithmetic operator minus.
   friend inline RationalTBase< UnderlingCRI > operator -( const RationalTBase< UnderlingCRI >& rMinuend, const RationalTBase<
UnderlingCRI >& rSubtrahend )
   {
      return( rMinuend + (-rSubtrahend) );
   }
   // Standard arithmetic operator minus.
   friend inline RationalTBase< UnderlingCRI > operator -( const Integer& iMinuend, const RationalTBase< UnderlingCRI >& rSubtrahend
)
   {
      return( RationalTBase< UnderlingCRI >( specialize< UnderlingCRI as sX::Numerator >::Initialize( iMinuend ), specialize<
UnderlingCRI as sX::Denominator >::Initialize( 1 ) ) + (-rSubtrahend) );
   }


   // Standard arithmetic operator minus.
   friend inline RationalTBase< UnderlingCRI > operator -( const RationalTBase< UnderlingCRI >& rMinuend, const Integer& iSubtrahend
)
   {
      return( rMinuend + (-RationalTBase< UnderlingCRI >( specialize< UnderlingCRI as sX::Numerator >::Initialize( iSubtrahend ),
specialize< UnderlingCRI as sX::Denominator >::Initialize( 1 ) )) );
   }


   // Standard arithmetic operator multiplication.
   friend inline RationalTBase< UnderlingCRI > operator *( const RationalTBase< UnderlingCRI >& rFactorL, const RationalTBase<
UnderlingCRI >& rFactorR )
   {
      return( Rational( rFactorL.bNegative ^ rFactorR.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize(
rFactorL.uiN * rFactorR.uiN ), specialize< UnderlingCRI as sX::Denominator >::Initialize( rFactorL.uiD * rFactorR.uiD ) ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline RationalTBase< UnderlingCRI > operator *( const RationalTBase< UnderlingCRI >& rFactorL, const Integer& iFactorR )
   {
      return( Rational( rFactorL.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize( iFactorR * rFactorL.uiN ),
specialize< UnderlingCRI as sX::Denominator >::Initialize( rFactorL.uiD ) ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline RationalTBase< UnderlingCRI > operator *( const Integer& iFactorL, const RationalTBase< UnderlingCRI >& rFactorR )
   {
      return( Rational( rFactorR.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize( iFactorL * rFactorR.uiN ),
specialize< UnderlingCRI as sX::Denominator >::Initialize( rFactorR.uiD ) ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline RationalTBase< UnderlingCRI > operator *( const RationalTBase< UnderlingCRI >& rFactorL, const unsigned __int64&
iFactorR )
   {
      return( Rational( rFactorL.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize( Integer( iFactorR ) *
rFactorR.uiN ), specialize< UnderlingCRI as sX::Denominator >::Initialize( rFactorL.uiD ) ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline RationalTBase< UnderlingCRI > operator *( const unsigned __int64& iFactorL, const RationalTBase< UnderlingCRI >&
rFactorR )
   {
      return( Rational( rFactorR.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize( Integer( iFactorL ) *
rFactorR.uiN ), specialize< UnderlingCRI as sX::Denominator >::Initialize( rFactorR.uiD ) ) );
   }


   // Standard arithmetic operator division. This operator throws
   // NumberException exception when dividing by zero.
   friend inline RationalTBase< UnderlingCRI > operator /( const RationalTBase< UnderlingCRI >& rDividend, const RationalTBase<
UnderlingCRI >& rDivisor )
   {
      NumberException< RationalTBase< UnderlingCRI >, true >::TestDivisionByZero( rDivisor );

      return( Rational( rDividend.bNegative ^ rDivisor.bNegative, specialize< UnderlingCRI as sX::Numerator >::Initialize(
rDividend.uiN * rDivisor.uiD ), specialize< UnderlingCRI as sX::Denominator >::Initialize( rDividend.uiD * rDivisor.uiN ) ) );
   }
};


typedef RationalTBase< Integer > Rational;
```

4.2.3. The Quadratic Irrational Template Class

The quadratic irrational template represents Real numbers in the form of quadratic irrational and similar to them. The class consists from a Rational and a Functional. The functional is a template on its own specialized with the specializer of the quadratic template. The quadratic specializer is propagated onto the functional and is not otherwise used by the quadratic type. The functional expects to be specialized with a class derived from an abstract class called Functor. The Functor represents a real function with a rational variable. The functor is defined as abstract class as follows:

$$Functor(x) := f\left(x\right),\ where\, x \in \mathbb{Q},\ f\, is\, a\, Real\, function.$$

The Functional is defined as:

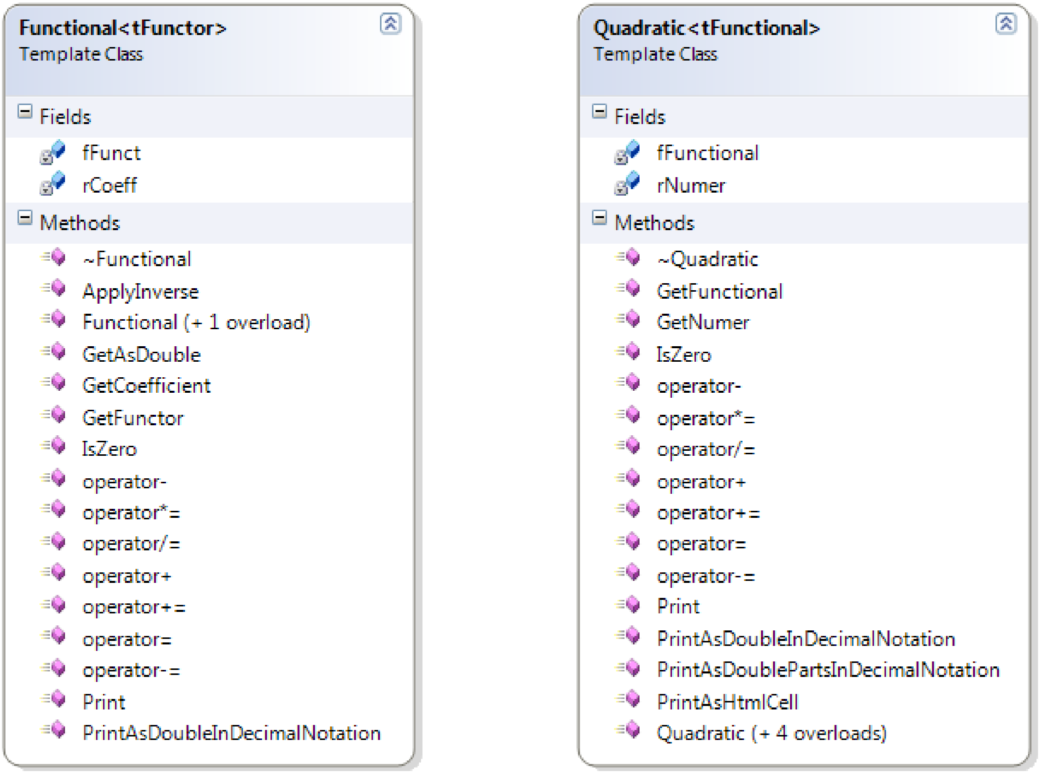
$$Functional(Functor(x),\ y) := yFunctor(x),\ where\, y \in \mathbb{Q}.$$

The Quadratic is defined as:

$$Quadratic(Functional(Functor(x),\ y),\ z) := z + Functional(Functor(x),\ y),\ where\, z \in \mathbb{Q}.$$

Thus the definition of quadratic number is $q = z + yf(x)$, where $x,\ y,\ z \in \mathbb{Q}$ and f is an unspecisied real function with inverse. In particular f could be defined as $f := \sqrt{x},\ f := \sqrt[3]{x}$, or any other real function with one variable.

Figure 11. Class diagram of the Functional and Quadratic template classes.



Functional<tFunctor>
Template Class

Fields
- fFunct
- rCoeff

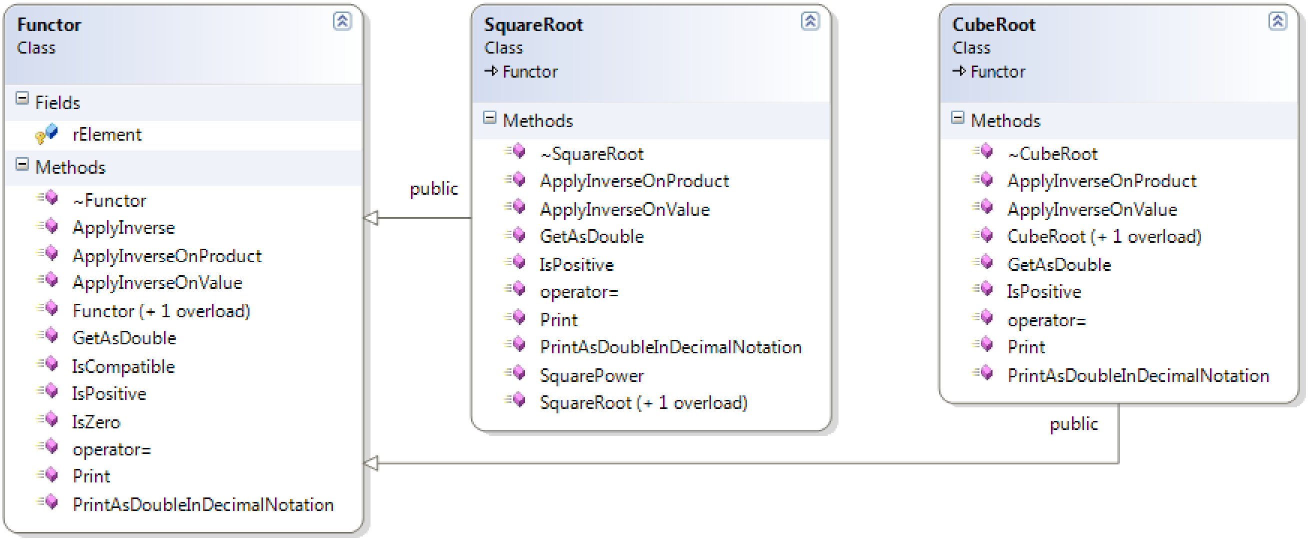
Methods
- ~Functional
- ApplyInverse
- Functional (+ 1 overload)
- GetAsDouble
- GetCoefficient
- GetFunctor
- IsZero
- operator-
- operator*=
- operator/=
- operator+
- operator+=
- operator=
- operator-=
- Print
- PrintAsDoubleInDecimalNotation

Quadratic<tFunctional>
Template Class

Fields
- fFunctional
- rNumer

Methods
- ~Quadratic
- GetFunctional
- GetNumer
- IsZero
- operator-
- operator*=
- operator/=
- operator+
- operator+=
- operator=
- operator-=
- Print
- PrintAsDoubleInDecimalNotation
- PrintAsDoublePartsInDecimalNotation
- PrintAsHtmlCell
- Quadratic (+ 4 overloads)

Figure 12. Class diagram of the Functor abstract class and two derived from it functors.



Although the template class is called Quadratic it is clear that in fact the it is quadratic if oand only if the specializer that is used to specialize the template is a square root functor. To define an instance of quadratic one need to use the following definition:

```
Rational a1, a2, a3, b1, b2, b3;

// Initialize a1, a2, a3, b1, b2, b3;

Quadratic< SquareRoot > q1( a1, Functional< SquareRoot >( a2, SquareRoot( a3 ) ) );
Quadratic< SquareRoot > q2( b1, Functional< SquareRoot >( b2, SquareRoot( b3 ) ) );
Quadratic< SquareRoot > q3( q1 + 208 * q2 );

// Define other cube-root, quadratic form irrational numver.
Quadratic< CubeRoot > c1( a1, Functional< CubeRoot >( a2, CubeRoot( a3 ) ) );
Quadratic< CubeRoot > c2( b1, Functional< CubeRoot >( b2, CubeRoot( b3 ) ) );
Quadratic< CubeRoot > c3( q1 + 208 * q2 );
```

Problem 27. The Quadratic template definition appears consistent; however it is impossible to have default constructor on the Quadratic since the variable under the functor cannot be predefined. Thus one or more of the following four is true:

1. The declaration above meets the requirement and represents the required irrational numbers however it is inconsistent in general, i.e. non bijective.
2. It is not possible to have more than one under-functor-variable per universe.
3. Default constructor is not necessary to exist.
4. There is a fundamental problem with quadratic numbers definition or numbers in general.

Clearly this problem is deep and serious and requires an investigation on its own. The Quadratic template as defined above is sufficiently consistent for the Mean Median Map investigation and was successfully used in it with this present form however the problem is extremely interesting and deserves the appropriate attention.

Listing 8. The Quadratic and Functional templates, and the Functor, SquareRoot and CubeRoot classes.

```
//
//
// Quadratic.h: Quadratic with arbitrary precision.
//
//                Proper Numbers Library
//
// © Copyright 2011 - 2011 by Miroslav Bonchev Bonchev. All rights reserved.
//
//
//*******************************************************************************************************


// Open Source License – The MIT License
//
//
// {your product} uses the: Proper Numbers Library – Copyright © 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
// {your product} uses the: Object Specialization Model – Copyright © 2009 - 2011 by Miroslav Bonchev Bonchev. All Rights Reserved.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
// associated  documentation files  (the "Software"),  to deal  in the Software without restriction,
// including  without  limitation the rights  to use,  copy,  modify,  merge,  publish,  distribute,
// sublicense,  and/or sell copies of the Software,  and to permit persons to  whom the  Software is
// furnished to do so, subject to the following conditions:
//
// The  above  copyright  notice  and  this  permission  notice  shall be  included in all copies or
// substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT
// NOT  LIMITED  TO  THE  WARRANTIES  OF  MERCHANTABILITY,  FITNESS  FOR  A  PARTICULAR  PURPOSE AND
// NONINFRINGEMENT.  IN NO EVENT SHALL  THE  AUTHORS  OR  COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
// DAMAGES OR OTHER LIABILITY,  WHETHER IN AN ACTION OF CONTRACT,  TORT OR OTHERWISE,  ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.


//___________________________________________________________________________________________________
// This software is OSI Certified Open Source Software.
// OSI Certified is a certification mark of the Open Source Initiative.


#pragma once


#include "stdafx.h"
#include "Common.h"
#include "StringEx.h"
#include "Specialize.h"
#include "Rational.h"


#define Minus_Char_TextSize         L"22"
#define Rational_Notation_TextSize  L"12"
#define Decimal_Notation_TextSize   L"12"


// Functor is a function that remains as it is e.g. SRQT( 5 )
class Functor
{
protected:
   Rational rElement;

public:
   Functor( const Rational& rElement ) : rElement( rElement )
   {
   }

   Functor( const Functor& function ) : rElement( function.rElement )
   {
   }

   virtual ~Functor()
   {
   }

   Functor& operator=( const Functor& function )
   {
      rElement = function.rElement;

      return( *this );
   }

   virtual bool IsZero() const
   {
      return( rElement.IsZero() );
   }


   virtual MStringEx< TCHAR > Print() const = 0;
   virtual MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const = 0;


   virtual Rational ApplyInverse() const
   {
      return( rElement );
   }


   virtual Rational ApplyInverseOnValue( const Rational& rNumber ) const = 0;
   virtual Rational ApplyInverseOnProduct( const Rational& rNumber ) const = 0;
   virtual double   GetAsDouble() const = 0;


   virtual bool IsPositive() const = 0;

   bool IsCompatible( const Functor& functor ) const
   {
      return( rElement == functor.rElement );
   }


   friend inline bool operator ==( const Functor& rLeft, const Functor& rRight )
   {
      return( rLeft.rElement == rRight.rElement );
   }


   friend inline bool operator !=( const Functor& rLeft, const Functor& rRight )
   {
      return( ! operator ==( rLeft, rRight ) );
   }
};


class SquareRoot : public Functor
{
public:
   SquareRoot( const Rational& rational ) : Functor( rational )
   {
   }


   SquareRoot( const SquareRoot& squareRoot ) : Functor( squareRoot )
   {
   }


   virtual ~SquareRoot()
   {
   }


   SquareRoot& operator=( const SquareRoot& squareRoot )
   {
      __super::operator=( squareRoot );

      return( *this );
   }


   virtual MStringEx< TCHAR > Print() const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("sqrt( %s )"), (LPCTSTR)rElement.Print() ) );
   }


   virtual MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("sqrt( %s )"), (LPCTSTR)rElement.PrintAsDoubleInDecimalNotation(
b1Precision ) ) );
```

```
   }


   virtual bool IsPositive() const
   {
      return( true );
   }


   const Rational& SquarePower() const
   {
      return( rElement );
   }


   virtual Rational ApplyInverseOnValue( const Rational& rNumber ) const
   {
      return( rNumber.ToPower( 2 ) );
   }


   virtual Rational ApplyInverseOnProduct( const Rational& rNumber ) const
   {
      return( rNumber.ToPower( 2 ) * ApplyInverse() );
   }


   virtual double GetAsDouble() const
   {
      return( sqrt( rElement.GetAsDouble() ) );
   }
};


class CubeRoot : public Functor
{
public:
   CubeRoot( const Rational& rational ) : Functor( rational )
   {
   }


   CubeRoot( const CubeRoot& cubeRoot ) : Functor( cubeRoot )
   {
   }


   virtual ~CubeRoot()
   {
   }


   CubeRoot& operator=( const CubeRoot& squareRoot )
   {
      __super::operator=( squareRoot );

      return( *this );
   }


   virtual MStringEx< TCHAR > Print() const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("cbrt( %s )"), (LPCTSTR)rElement.Print() ) );
   }


   virtual MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("cbrt( %s )"), (LPCTSTR)rElement.PrintAsDoubleInDecimalNotation(
b1Precision ) ) );
   }


   virtual bool IsPositive() const
   {
      return( true );
   }


   virtual Rational ApplyInverseOnValue( const Rational& rNumber ) const
   {
      return( rNumber.ToPower( 3 ) );
   }


   virtual Rational ApplyInverseOnProduct( const Rational& rNumber ) const
   {
      return( rNumber.ToPower( 3 ) * ApplyInverse() );
   }


   virtual double GetAsDouble() const
   {
      return( pow( rElement.GetAsDouble(), (double)1 / (double)3 ) );
   }
};


// r1*functor(r2)
template< class tFunctor >
class Functional
{
private:
   Rational rCoeff;
   tFunctor fFunct;


public:
   // Copy constructor.
   Functional( const Functional& functional ) : rCoeff( functional.rCoeff ), fFunct( functional.fFunct )
   {
   }


   // Constructor from unsigned numerator and denominator.
   Functional( const Rational& rCoeff, const tFunctor& fFunct ) : rCoeff( rCoeff ), fFunct( fFunct )
   {
   }


   // Destructor.
   ~Functional()
   {
   }


   // Standard operator equal.
   Functional& operator=( const Functional& functional )
   {
      rCoeff = functional.rCoeff;
      fFunct = functional.fFunct;

      return( *this );
   }


   // Returns a double approximation of the functional value.
   double GetAsDouble() const
   {
      return( rCoeff.GetAsDouble() * fFunct.GetAsDouble() );
   }


   Rational ApplyInverse() const
   {
      return( fFunct.ApplyInverseOnProduct( rCoeff ) );
   }


   // Standard operator +=
   Functional< tFunctor > operator +=( const Functional< tFunctor >& fSummand )
   {
      return( *this = *this + fSummand );
   }


   // Standard operator -=
   Functional< tFunctor > operator -=( const Functional< tFunctor >& fSubtrahend )
   {
      return( *this = *this - fSubtrahend );
   }


   // Standard operator *=
   Functional< tFunctor > operator *=( const Functional< tFunctor >& rFactor )
   {
      return( *this = *this * rFactor );
   }


   // Standard operator /=
   Functional< tFunctor > operator /=( const Functional< tFunctor >& rDivisor )
   {
      return( *this = *this / rDivisor );
   }


   // Unary plus (returns the operand itself).
   Functional< tFunctor > operator+() const
   {
      return( *this );
   }


   // Unary operator negation - inverts the signd.
   Functional< tFunctor > operator-() const
   {
      return( Functional< tFunctor >( rCoeff, fFunct ) );
   }


   // Returns the coefficient.
   const Rational& GetCoefficient() const
   {
      return( rCoeff );
   }


   // Returns the functor.
   const tFunctor& GetFunctor() const
   {
      return( fFunct );
   }


   // Test if the number is zero.
   bool IsZero() const
   {
      return( rCoeff.IsZero() || fFunct.IsZero() );
   }


   // Print the Functional exactly.
   MStringEx< TCHAR > Print() const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%s.%s"), (LPCTSTR)rCoeff.Print(), (LPCTSTR)fFunct.Print() ) );
   }


   // This prints the Functional using double representation of the contained numbers.
   // The parameter sets the required number of digits after the decimal point, max number is 16. Note that
   // the floating point representation is most likely inaccurate to some degree depending on the represented
   // number. Floating point numbers cannot represent proper rational numbers satisfactory.
   MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%s.%s"), (LPCTSTR)rCoeff.PrintAsDoubleInDecimalNotation( b1Precision ),
(LPCTSTR)fFunct.PrintAsDoubleInDecimalNotation( b1Precision ) ) );
   }


   // External operators - friend operators.


   // Standard comparison operator equal to.
   friend inline bool operator ==( const Functional< tFunctor >& fLeft, const Functional< tFunctor >& fRight )
   {
      NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

      return( fLeft.rCoeff == fRight.rCoeff );
   }
```

```
// Standard comparison operator not equal to.
friend inline bool operator !=( const Functional< tFunctor >& rLeft, const Functional< tFunctor >& rRight )
{
   NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

   return( fLeft.rCoeff != fRight.rCoeff );
}


// Standard comparison operator less than.
friend inline bool operator <( const Functional< tFunctor >& rLeft, const Functional< tFunctor >& rRight )
{
   NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

   return( fLeft.rCoeff < fRight.rCoeff );
}


// Standard comparison operator greater than.
friend inline bool operator >( const Functional< tFunctor >& rLeft, const Functional< tFunctor >& rRight )
{
   NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

   return( fLeft.rCoeff > fRight.rCoeff );
}


// Standard comparison operator less than or equal to.
friend inline bool operator <=( const Functional< tFunctor >& rLeft, const Functional< tFunctor >& rRight )
{
   NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

   return( fLeft.rCoeff <= fRight.rCoeff );
}


// Standard comparison operator greater than or equal to.
friend inline bool operator >=( const Functional< tFunctor >& rLeft, const Functional< tFunctor >& rRight )
{
   NumberException< tFunctional, true >::TestCompatibility( fLeft.fFunct, fRight.fFunct );

   return( fLeft.rCoeff >= fRight.rCoeff );
}


// Standard arithmetic operator plus.
friend inline Functional< tFunctor > operator +( const Functional< tFunctor >& fSummandL, const Functional< tFunctor >& fSummandR )
{
   NumberException< tFunctor, true >::TestCompatibility( fSummandL.fFunct, fSummandR.fFunct );

   // r1*f(r) + r2*f(r) = (r1 + r2)*f(r)
   return( Functional< tFunctor >( fSummandL.rCoeff + fSummandR.rCoeff, fSummandL.fFunct ) );
}


// Standard arithmetic operator minus.
friend inline Functional< tFunctor > operator -( const Functional< tFunctor >& fMinuend, const Functional< tFunctor >& fSubtrahend )
{
   NumberException< tFunctor, true >::TestCompatibility( fMinuend.fFunct, fSubtrahend.fFunct );

   // r1*f(r) - r2*f(r) = (r1 + r2)*f(r)
   return( Functional< tFunctor >( fMinuend.rCoeff - fSubtrahend.rCoeff, fMinuend.fFunct ) );
}


// Arithmetic operator multiplication - Rational = Functional * Functional
// Big problems here. This operator must return a Functional IN GENERAL. The Coefficient part has no problems,
// But the functor part presents a large proplem, it needs to change so I need to return VARIANT type...
friend inline Rational operator *( const Functional< tFunctor >& fFactorL, const Functional< tFunctor >& fFactorR )
{
   NumberException< tFunctor, true >::TestCompatibility( fFactorL.fFunct, fFactorR.fFunct );

   return( fFactorL.rCoeff * fFactorR.rCoeff * fFactorL.fFunct.SquarePower() );
}


// Standard arithmetic operator multiplication.
friend inline Functional< tFunctor > operator *( const Rational& rLeft, const Functional< tFunctor >& fFactorR )
```

```
   {
      return( Functional< tFunctor >( rLeft * fFactorR.rCoeff, fFactorR.fFunct ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline Functional< tFunctor > operator *( const Functional< tFunctor >& fFactorL, const Rational& rRight )
   {
      return( Functional< tFunctor >( rRight * fFactorL.rCoeff, fFactorL.fFunct ) );
   }


   // Standard arithmetic operator division. This operator throws
   // NumberException exception when dividing by zero.
   friend inline Functional< tFunctor > operator /( const Functional< tFunctor >& rDividend, const Functional< tFunctor >& rDivisor )
   {
      NumberException< Functional< tFunctor >, true >::TestDivisionByZero( rDivisor );
      NumberException< tFunctor, true >::TestCompatibility( rDividend.fFunct, rDivisor.fFunct );

      return( Functional< tFunctor >( rDividend.rCoeff / rDivisor.rCoeff, rDividend.fFunct / rDivisor.fFunct ) );
   }
};


template< class tFunctional >
class Quadratic //TBase
{
private:
   Rational    rNumer;
   Functional< tFunctional > fFunctional;


public:
   // Copy constructor.
   Quadratic( const Quadratic< tFunctional >& quadratic )
      :  rNumer( quadratic.rNumer ),
         fFunctional( quadratic.fFunctional )
   {
   }


   // Constructor from signed integer.
   Quadratic( const __int64 nInteger )
      :  rNumer( nInteger ),
         fFunctional( 0, tFunctional( 5 ) )
   {
   }


   // Constructor from integer.
   Quadratic( const Integer& iNumer )
      :  rNumer( specialize< Integer as sX::Numerator >::Initialize( rNumer ),
         fFunctional( 0, tFunctional( 5 ) ) )
   {
   }


   // Constructor from Rational.
   Quadratic( const Rational& rNumer )
      :  rNumer( rNumer ),
         fFunctional( 0, tFunctional( 5 ) )
   {
   }


   // Constructor from Rational and functional.
   Quadratic( const Rational& rNumer, const Functional< tFunctional >& fFunctional )
      :  rNumer( rNumer ),
         fFunctional( fFunctional )
   {
   }


   // Destructor.
   ~Quadratic()
   {
   }


   // Standard operator equal.
   Quadratic< tFunctional >& operator=( const Quadratic< tFunctional >& quadratic )
   {
      rNumer = quadratic.rNumer;
      fFunctional = quadratic.fFunctional ;

      return( *this );
   }


   // Standard operator +=
   Quadratic< tFunctional > operator +=( const Quadratic< tFunctional >& rSummand )
   {
      return( *this = *this + rSummand );
   }


   // Standard operator -=
   Quadratic< tFunctional > operator -=( const Quadratic< tFunctional >& rSubtrahend )
   {
      return( *this = *this - rSubtrahend );
   }


   // Standard operator *=
   Quadratic< tFunctional > operator *=( const Quadratic< tFunctional >& rFactor )
   {
      return( *this = *this * rFactor );
   }


   // Standard operator /=
   Quadratic< tFunctional > operator /=( const Quadratic< tFunctional >& rDivisor )
   {
      return( *this = *this / rDivisor );
   }


   // Unary plus (returns the operand itself).
   Quadratic< tFunctional > operator+() const
   {
      return( *this );
   }


   // Unary operator negation – inverts the sign.
   Quadratic< tFunctional > operator-() const
   {
      return( Quadratic< tFunctional >( -rNumer, -fFunctional ) );
   }


   // Test if the number is zero.
   bool IsZero() const
   {
      return( rNumer.IsZero() && fFunctional.IsZero() );
   }


   // This method prints a string exact representations of the quadratic number.
   // This method is intended for use when the Rational is specialized with the Integer class. If used with
   // another specializer type, it must have method string Print() const, where string must have LPCTSTR
   // overloaded operator. Although this method returns the exact rational number, in some circumstances
   // one may want to also consider using the float representations for informative purposes as their results
   // may be easier to interpret.
   MStringEx< TCHAR > Print() const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%s %s"), (LPCTSTR)rNumer.Print(), (LPCTSTR)fFunctional.Print() ) );
   }


   // This method returns the double representation of the rational number and functional as a string in decimal notation.
   // The parameter sets the required number of digits after the decimal point, max number is 16. Note that
   // the floating point representation is most likely inaccurate to some degree depending on the represented
   // number. Floating point numbers cannot represent proper rational numbers satisfactory.
   MStringEx< TCHAR > PrintAsDoublePartsInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%s %s"), (LPCTSTR)rNumer.PrintAsDoubleInDecimalNotation( b1Precision ),
(LPCTSTR)fFunctional.PrintAsDoubleInDecimalNotation( b1Precision ) ) );
   }
   // This method returns the double approximation of the number.
   // The parameter sets the required number of digits after the decimal point, max number is 16. Note that
   // the floating point representation is most likely inaccurate to some degree depending on the represented
   // number. Floating point numbers cannot represent proper rational numbers satisfactory.
   MStringEx< TCHAR > PrintAsDoubleInDecimalNotation( const BYTE b1Precision ) const
   {
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF, MStringEx< TCHAR >( MStringEx< TCHAR >::FF, TEXT("%%.%df"), b1Precision ),
rNumer.GetAsDouble() + fFunctional.GetAsDouble() ) );
   }


   // This method returns a HTML table containing the Quadratic number in floating point notation. This
   // method is intended for use when the Rational is specialized with the Integer class. If used with
   // another specializer type, it must have method double GetAsDouble() const. Note that a number from
   // the Rational class may be much larger than the larger number that a double precision floating point
   // could represent and is also PRECISE. Also, note that floating point numbers are not evenly
   // distributed having irregularly sized gaps between them. In difference the Rational specialized
   // with Integer are evenly distributed and PRECISE. This means that Rational specialized with Integer
   // number could be in fact very rarely represented using a floating point number. This function is
   // useful for displaying Rational numbers for informative purposes.
   MStringEx< TCHAR > PrintAsHtmlCell() const
   {
      // |- table -------------------|-----------------------|
      // |                quadratic  |  double approximation |
      // |---------------------------|-----------------------|
      return( MStringEx< TCHAR >( MStringEx< TCHAR >::FF,
                                  L"<table border='0' cellpadding='0' cellspacing='0' style='text-align: right;'>\
                                      <tr>\
                                        <td style='font-size: 12px; width:500px; text-align: right; font-size:" Decimal_Notation_TextSize
L"px; width: 300px;'>%s</td>\
                                        <td style='padding-left:20px; text-align: left; font-size:" Decimal_Notation_TextSize L"px; width:
200px;'>%s</td>\
                                      </tr>\
                                    </table>", (LPCTSTR)Print(), (LPCTSTR)PrintAsDoubleInDecimalNotation( 16 ) ) );
   }


   // External operators - friend operators.


   // Standard comparison operator equal to.
   friend inline bool operator ==( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      NumberException< tFunctional, true >::TestCompatibility( rLeft.fFunctional.GetFunctor(), rRight.fFunctional.GetFunctor() );

      // | a1     c1  |   | a2     c2  |    | a1     a2 |   | c2     c1 |
      // |---- + ----m| ? |---- + ----m| => |---- - ----| ? |---- - ----|m => (x1 -x2) ? (y2-y1)m
      // | b1     d1  |   | b2     d2  |    | b1     b2 |   | d2     d1 |

      return( rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( rLeft.rNumer - rRight.rNumer ) == Functional< tFunctional >(
rRight.fFunctional.GetCoefficient() - rLeft.fFunctional.GetCoefficient(), rLeft.fFunctional.GetFunctor() ).ApplyInverse() );
   }


   // Standard comparison operator not equal to.
   friend inline bool operator !=( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      return( !operator==( rLeft, rRight ) );
   }


   // Standard comparison operator less than.
   friend inline bool operator <( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      NumberException< tFunctional, true >::TestCompatibility( rLeft.fFunctional.GetFunctor(), rRight.fFunctional.GetFunctor() );

      // | a1     c1  |   | a2     c2  |    | a1     a2 |   | c2     c1 |
      // |---- + ----m| < |---- + ----m| -> |---- - ----| < |---- - ----|m -> (x1 -x2) < (y2-y1)m -> X < Ym
      // | b1     d1  |   | b2     d2  |    | b1     b2 |   | d2     d1 |

      const Rational rX( rLeft.rNumer - rRight.rNumer );
      const Rational rY( rRight.fFunctional.GetCoefficient() - rLeft.fFunctional.GetCoefficient() );

      const bool bX_positive( 0 < rX );
      const bool bYm_positive( (0 < rY) && rRight.fFunctional.GetFunctor().IsPositive() );

      if( bX_positive != bYm_positive )
      {
         // One side is positive, the other negative. rYm > 0 => rYm > rX.
         return( bYm_positive );
      }

      // Both sides are of the same sign.
      if( bX_positive || (rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( -1 ) < 0) )
      {
         // Both sides are positive
         // OR the inverse of the functor does not change the sign - consider m is CubeRoot.
         // Compare the inverses as it is.
         return( rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( rX ) < Functional< tFunctional >( rY, rLeft.fFunctional.GetFunctor()
).ApplyInverse() );
      }

      // Both sides are negative, and the inverse changes the sign - consider m is SquareRoot.
      return( rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( rX ) > Functional< tFunctional >( rY, rLeft.fFunctional.GetFunctor()
).ApplyInverse() );
   }


   // Standard comparison operator greater than.
   friend inline bool operator >( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      NumberException< tFunctional, true >::TestCompatibility( rLeft.fFunctional.GetFunctor(), rRight.fFunctional.GetFunctor() );

      // | a1     c1  |   | a2     c2  |    | a1     a2 |   | c2     c1 |
      // |---- + ----m| > |---- + ----m| -> |---- - ----| > |---- - ----|m -> (x1 -x2) > (y2-y1)m -> X > Ym
      // | b1     d1  |   | b2     d2  |    | b1     b2 |   | d2     d1 |

      const Rational rX( rLeft.rNumer - rRight.rNumer );
      const Rational rY( rRight.fFunctional.GetCoefficient() - rLeft.fFunctional.GetCoefficient() );

      const bool bX_positive( 0 < rX );
      const bool bYm_positive( (0 < rY) && rRight.fFunctional.GetFunctor().IsPositive() );

      if( bX_positive != bYm_positive )
      {
         // One side is positive, the other negative. rX > 0 => rX > rYm.
         return( bX_positive );
      }

      // Both sides are of the same sign.
      if( bX_positive || (rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( -1 ) < 0) )
      {
         // Both sides are positive
         // OR the inverse of the functor does not change the sign - consider m is CubeRoot.
         // Compare the inverses as it is.
         return( rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( rX ) > Functional< tFunctional >( rY, rLeft.fFunctional.GetFunctor()
).ApplyInverse() );
      }

      // Both sides are negative, and the inverse changes the sign - consider m is SquareRoot.
      return( rLeft.fFunctional.GetFunctor().ApplyInverseOnValue( rX ) < Functional< tFunctional >( rY, rLeft.fFunctional.GetFunctor()
).ApplyInverse() );
   }


   // Standard comparison operator less than or equal to.
   friend inline bool operator <=( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      return( operator< ( rLeft, rRight ) || operator==( rLeft, rRight ) );
   }


   // Standard comparison operator greater than or equal to.
   friend inline bool operator >=( const Quadratic< tFunctional >& rLeft, const Quadratic< tFunctional >& rRight )
   {
      return( operator> ( rLeft, rRight ) || operator==( rLeft, rRight ) );
   }


   // Standard arithmetic operator plus.
   friend inline Quadratic< tFunctional > operator +( const Quadratic< tFunctional >& iSummandL, const Quadratic< tFunctional >& iSummandR
)
   {
      return( Quadratic< tFunctional >( iSummandL.rNumer + iSummandR.rNumer, iSummandL.fFunctional + iSummandR.fFunctional ) );
   }


   // Standard arithmetic operator minus.
   friend inline Quadratic< tFunctional > operator -( const Quadratic< tFunctional >& qMinuend, const Quadratic< tFunctional >& 
qSubtrahend )
   {
      return( Quadratic< tFunctional >( qMinuend.rNumer - qSubtrahend.rNumer, qMinuend.fFunctional - qSubtrahend.fFunctional ) );
   }


   // Standard arithmetic operator multiplication.
   friend inline Quadratic< tFunctional > operator *( const Quadratic< tFunctional >& qFactorL, const Quadratic< tFunctional >& qFactorR )
   {
      return( Quadratic< tFunctional >( qFactorL.rNumer * qFactorR.rNumer + qFactorL.fFunctional * qFactorR.fFunctional, qFactorL.rNumer * 
qFactorR.fFunctional + qFactorL.fFunctional * qFactorR.rNumer ) );
   }


   // Standard arithmetic operator division. This operator throws
   // NumberException exception when dividing by zero.
   friend inline Quadratic< tFunctional > operator /( const Quadratic< tFunctional >& qDividend, const Quadratic< tFunctional >& qDivisor 
)
   {
      NumberException< Quadratic< tFunctional >, true >::TestDivisionByZero( qDivisor );
      NumberException< tFunctional, true >::TestCompatibility( qDividend.fFunctional.GetFunctor(), qDivisor.fFunctional.GetFunctor() );

      // | a1     c1  |   | a2     c2 |        b2.d2                  mD^2
      // |---- + ----m| / |---- + ----| = A = ------- . --------------------------------- .[ B = (a1.a2.d1.d2 - b1.b2.c1.c2.m^2) + C = 
(a2.b1.c1.d2 - a1.b2.c2.d1).m]
      // | b1     d1  |   | b2     d2 |        b1.d1     (a2.d2)^2.mD^2 - (b2.c2)^2.mN^2

      const Rational a1( qDividend.rNumer.IsNegative(), specialize< Integer as sX::Numerator >::Initialize( 
qDividend.rNumer.GetNumerator() ) );
      const Rational b1( specialize< Integer as sX::Numerator >::Initialize( qDividend.rNumer.GetDenominator() ) );

      const Rational c1( qDividend.fFunctional.GetCoefficient().IsNegative(), specialize< Integer as sX::Numerator >::Initialize( 
qDividend.fFunctional.GetCoefficient().GetNumerator() ) );
      const Rational d1( specialize< Integer as sX::Numerator >::Initialize( qDividend.fFunctional.GetCoefficient().GetDenominator() ) );


      const Rational a2( qDivisor.rNumer.IsNegative(), specialize< Integer as sX::Numerator >::Initialize( qDivisor.rNumer.GetNumerator() 
) );
      const Rational b2( specialize< Integer as sX::Numerator >::Initialize( qDivisor.rNumer.GetDenominator() ) );

      const Rational c2( qDivisor.fFunctional.GetCoefficient().IsNegative(), specialize< Integer as sX::Numerator >::Initialize( 
qDivisor.fFunctional.GetCoefficient().GetNumerator() ) );
      const Rational d2( specialize< Integer as sX::Numerator >::Initialize( qDivisor.fFunctional.GetCoefficient().GetDenominator() ) );

      const Rational mSquare( qDividend.fFunctional.GetFunctor().SquarePower() );

      // Compute the singless parts.
      const Rational A( specialize< Integer as sX::Numerator >::Initialize( b2.GetNumerator() * d2.GetNumerator() * 
mSquare.GetDenominator() ),
                        specialize< Integer as sX::Denominator >::Initialize( b1.GetNumerator() * d1.GetNumerator() *
                                                                             (a2.GetNumerator() *
                                                                              a2.GetNumerator() *
                                                                              d2.GetNumerator() *
                                                                              d2.GetNumerator() *
                                                                              mSquare.GetDenominator() -
                                                                              c2.GetNumerator() *
                                                                              c2.GetNumerator() *
                                                                              b2.GetNumerator() *
                                                                              b2.GetNumerator() *
                                                                              mSquare.GetNumerator()) ) );

      const Rational B( a1 * a2 * d1 * d2 - b1 * b2 * c1 * c2 * mSquare );
      const Rational C( a2 * b1 * c1 * d2 - a1 * b2 * c2 * d1 );

      return( Quadratic< tFunctional >( A * B, Functional< tFunctional >( A * C, qDividend.fFunctional.GetFunctor() ) ) );
   }


   const Rational& GetNumer() const
   {
      return( rNumer );
   }


   const Functional< tFunctional >& GetFunctional() const
   {
      return( fFunctional );
   }
};
```

4.3. The Mean Meadin Map Application

The Mean Median Map application is a multithreaded console application able to run from 1 to 24 threads simultaneously. Since the Mean Median Map is very computational extensive it was designed to allow utilizing the entire machine processing power. The typical processor in a modern computer is multicore running at least one logical thread per core. Since the Mean Median Map is a recursive function it is not possible to be itself parallelized, however often one desires to compute the Mean Median Map for more than one starting set, e.g. on interval, by sequence (the starting sets are determined by some other well-known sequence, e.g. Fibonacci numbers), or otherwise. In these cases the application is able to launch and maintain threads up to the requested amount utilizing the processor resources. Suppose that one requires to compute large number of sequences on a system equipped with an i7-980x processor, which has 6 cores and 12 logical threads. Launching the application with request for use of 12 (work) threads will utilize all processor resources. Launching the application with 13 or more threads will cause performance degrade due threads switching. Launching 11 or fewer threads would be leaving resources unused. The work threads run at low priority level, thus even when they require 100% of the processor resources the operating system will schedule the user interface and other normal priority threads with precedence over them, so that the system is also usable for other tasks. Other systems and processors might have 1, 2, 4, 8 or other numbers of cores with one or more logical threads each. The maximum number of 24 work threads is selected as suitable but can be changed if desired to any other suitable number subject to system resources and support.

The Mean Median application has two builds controlled by a pre-processor definition “QUADRATIC” located on the top of MeanMedianMap.cpp as follows:

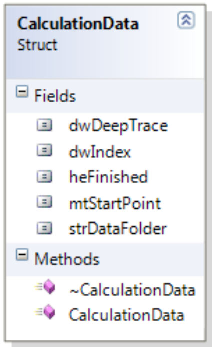
```
#define QUADRATIC 1

#ifdef QUADRATIC
   typedef Quadratic< SquareRoot > MEDIAN_TYPE;
#else
   typedef Rational MEDIAN_TYPE;
#endif
```

The data processing code is written using a type definition MEDIAN_TYPE, however the control of what exactly type the MEDIAN_TYPE is, is delegated to the "QUADRATIC” pre-processor switch. The reason is problem 27 which prohibits instantiation of Quadratic objects with default constructor and thus it becomes impossible for certain parts of the code to compile otherwise. There are other possible works around, however due time restraints the author chose this typical approach, maintaining the clarity of the code. A proper solution should be given after resolving problem 27. The QUADRATIC switch chooses between code compiled with Rational or Quadratic numbers. The type definition MEDIAN_TYPE allows the application to be extended in future with other than Rational and Quadratic types. Depending on the compilation Quadratic or Rational the Mean Median Map application have different command line options. The Rational numbers have effectively four different computation modes: equal-distant sequences on interval, single value, sequence in the form $\frac{2^{i-1}+1}{2^i}$, sequence of the Fibonacci numbers ratio $\frac{F_n}{F_{n+1}}$. The Quadratic numbers can compute only one sequence at a time. Should one desire they could expand these facilities and add other possible starting set definitions.

The Mean Median Map application records a general sequence information in file named #define DATA_FILE_NAME TEXT("\\Data.txt") and path entered at application launch. For each computed sequence the application creates a nested folder with name the index of the current sequence at the location given at start up with data recorded during the computation. Note that existing files are deleted without warning. Once the sequence is halted the application records the final X_n, X_n^{Sort}, M_n, E_n and $M_n - M_{n-1}$ in a file called Record_Final.html, a second file Record_Final.html.txt with the same content for machine use is also created at the same location. In addition for each halt kernels is recorded a separate files with appropriate name containing the halt kernel data. The software is able to also record all sequence data periodically after number of iterations subject of appropriate command line parameter. The later might be useful for slow halting sequences to allow observations on the sequence development.

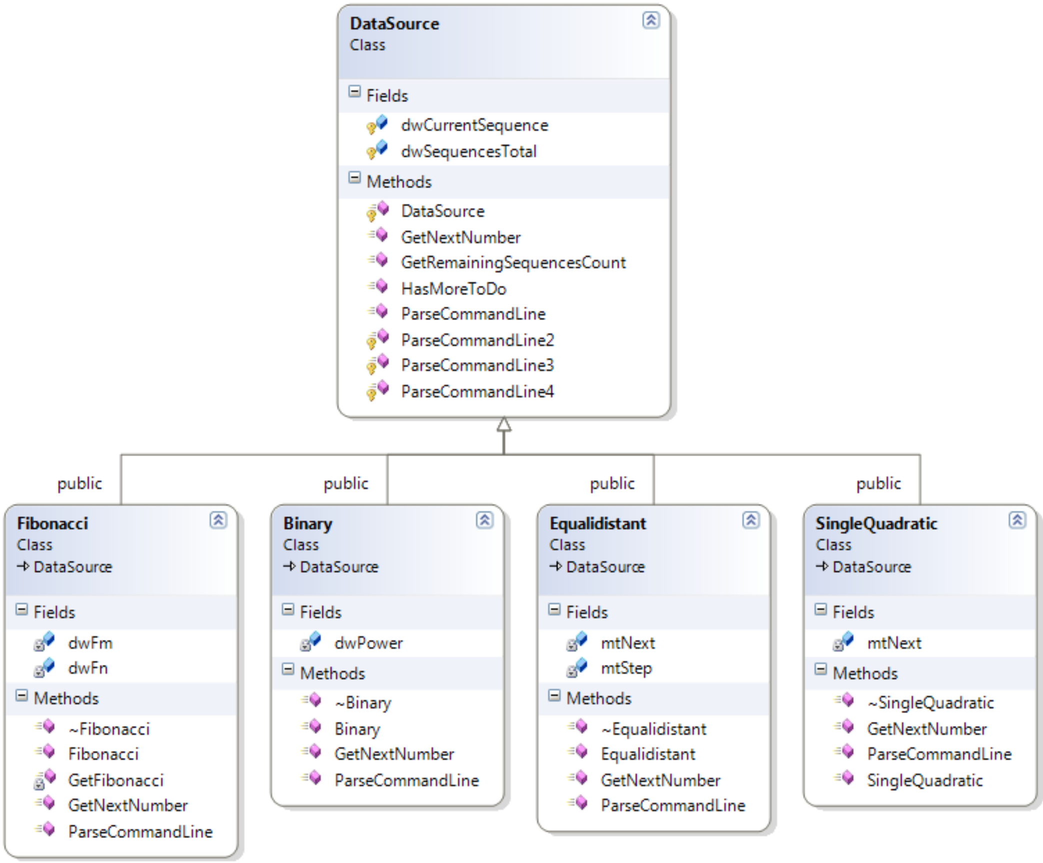
Architecturally the Mean Median Map application is divided in two parts - parser part and execution part. The parser part scans the command line to determine the requirements of the request. If the request is invalid it issues error, prints the parameters list and exits the application. If the request is valid, the parser processes it, creates a DataSource object and passes the DataSource object to the execution part. The execution part is essentially a loop managing the work threads. Upon available thread slot the execution part retrieves the next starting set from the DataSource object, spawns a new work thread and passes the starting set to it for processing. This process repeats until the user request is complete or the user terminates the application.

Figure 13. Work threads data transmission class diagram.



The Meam Median Map application uses the three points fact 5 and thus starting sets have the form $\{0,\, x,\, 1\}$, where $x \in \left[\frac{1}{2},\, 1\right]$. For this reason the DataSource object need to return only one number x.

Figure 14. Class diagram of the DataSource abstract class and the derived from it Data Sources.



Listing 9. The Mean Median Map Application.

```
//
//
// MeanMedianMap.cpp : Defines the entry point for the console application.
//
//                Mean Median Map sequences computation application
//
// © Copyright 2010 - 2011 by Miroslav Bonchev Bonchev. All rights reserved.
//
//
//*****************************************************************************************************


#include "stdafx.h"
#include "Common.h"
#include "MAtom.h"
#include "StringEx.h"
#include "Pair.h"
#include "Dictionary.h"
#include "Integer.h"
#include "Rational.h"
#include "Quadratic.h"
#include "Specialize.h"
#include <crtdbg.h>



// Declare Time and Space Domains
class Clock {};
class Value {};

// Declare the Xn and Mn sequences
class Xn {};
class Mn {};


#define POINT_0 0
#define POINT_1 1


#define MAX_COMPUTATION_THREAD_COUNT 24

#if MAX_COMPUTATION_THREAD_COUNT < 1
   #error The maximal thread count cannot be less than one.
#endif


#define DATA_FILE_NAME TEXT("\\Data.txt")


#define UniverseQuadraticFunctorType SquareRoot
#define UniverseQuadraticFunctorData SquareRoot( 5 )


#define QUADRATIC 1

#ifdef QUADRATIC
   typedef Quadratic< UniverseQuadraticFunctorType > MEDIAN_TYPE;
#else
   typedef Rational MEDIAN_TYPE;
#endif


class DataSource
{
protected:
   DWORD dwSequencesTotal;
   DWORD dwCurrentSequence;

   DataSource() : dwSequencesTotal( 0 ), dwCurrentSequence( 0 )
   {
   }

public:
   virtual MEDIAN_TYPE GetNextNumber() = 0;
   virtual bool HasMoreToDo() const
   {
      return( dwCurrentSequence < dwSequencesTotal );
   }


   virtual DWORD GetRemainingSequencesCount()
   {
      MASSERT( (0 != dwSequencesTotal) || (0 != dwCurrentSequence) );

      return( dwSequencesTotal - dwCurrentSequence );
   }

   virtual bool ParseCommandLine( _TCHAR* argv[], const DWORD dwArg ) = 0;

protected:
   virtual bool ParseCommandLine2( _TCHAR* argv[], const DWORD dwArg, DWORD& dwStartingIndex, DWORD& dwTotalSequences )
   {
      if( 7 != dwArg )
      {
         if( 7 < dwArg )
         {
            wprintf( TEXT("Invalid parameters call. Too many parameters.\n\n ") );
         }
         else
         {
            wprintf( TEXT("Invalid parameters call. Too few parameters.\n\n ") );
         }

         return( false );
      }


      // Check that the starting index is OK.
      dwStartingIndex = string( argv[5] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwStartingIndex ).CompareNoCase( argv[5] ) )
      {
         wprintf( TEXT("Invalid parameters call: The entered starting index could not be interpreted correctly. Please enter a decimal number from 0 to 2^32-1.\n\n ") );

         return( false );
      }


      // Check that the requested sequences is OK.
      dwTotalSequences = string( argv[6] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwTotalSequences ).CompareNoCase( argv[6] ) )
      {
         wprintf( TEXT("Invalid parameters call: The total sequences count could not be interpreted correctly. Please enter a decimal number from 1 to 2^32-1.\n\n ") );

         return( false );
      }


      if( 0 == dwTotalSequences )
      {
         wprintf( TEXT("Invalid parameters call: The total sequences cannot be zero. Please enter a decimal number from 1 to 2^32-1.\n\n ") );

         return( false );
      }

      return( true );
   }

   virtual bool ParseCommandLine4( _TCHAR* argv[], const DWORD dwArg, Rational& rStart, Rational& rEnd, DWORD& dwTotalSequences )
   {
      if( 9 != dwArg )
      {
         if( 9 < dwArg )
         {
            wprintf( TEXT("Invalid parameters call. Too many parameters.\n\n ") );
         }
         else
         {
            wprintf( TEXT("Invalid parameters call. Too few parameters.\n\n ") );
         }
         return( false );
      }


      // Check that the start numerator is OK.
      DWORD dwStartNumerator = string( argv[5] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwStartNumerator ).CompareNoCase( argv[5] ) )
      {
         wprintf( TEXT("Invalid parameters call: The entered start numerator could not be interpreted correctly. Please enter a
decimal number from 0 to 2^32-1.\n\n ") );

         return( false );
      }


      // Check that the end numerator is OK.
      DWORD dwEndNumerator = string( argv[6] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwEndNumerator ).CompareNoCase( argv[6] ) )
      {
         wprintf( TEXT("Invalid parameters call: The entered end numerator could not be interpreted correctly. Please enter a decimal
number from 0 to 2^32-1.\n\n ") );

         return( false );
      }


      if( dwStartNumerator > dwEndNumerator )
      {
         wprintf( TEXT("Invalid parameters call: The start numerator cannot be bigger than the end numerator. Please enter a decimal
number from 0 to 2^32-1.\n\n ") );

         return( false );
      }


      // Check that the end numerator is OK.
      DWORD dwDenominator = string( argv[7] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwDenominator ).CompareNoCase( argv[7] ) )
      {
         wprintf( TEXT("Invalid parameters call: The entered denominator could not be interpreted correctly. Please enter a decimal
number from 0 to 2^32-1.\n\n ") );

         return( false );
      }


      // Check that the requested sequences is OK.
      dwTotalSequences = string( argv[8] ).operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwTotalSequences ).CompareNoCase( argv[8] ) )
      {
         wprintf( TEXT("Invalid parameters call: The total sequences count could not be interpreted correctly. Please enter a decimal
number from 1 to 2^32-1.\n\n ") );

         return( false );
      }


      if( 0 == dwTotalSequences )
      {
         wprintf( TEXT("Invalid parameters call: The total sequences cannot be zero. Please enter a decimal number from 1 to 2^32-
1.\n\n ") );

         return( false );
      }


      if( (dwStartNumerator == dwEndNumerator) && (1 != dwTotalSequences) )
      {
         wprintf( TEXT("Invalid parameters call: The start and end numbers are the same. The total sequences cannot be other than
1.\n\n ") );

         return( false );
      }


      rStart = Rational( specialize< Integer as sX::Numerator >::Initialize( dwStartNumerator ), specialize< Integer as
sX::Denominator >::Initialize( dwDenominator ) );
```

```
      rEnd   = Rational( specialize< Integer as sX::Numerator >::Initialize( dwEndNumerator ), specialize< Integer as sX::Denominator
>::Initialize( dwDenominator ) );

      return( true );
   }


   virtual bool ParseCommandLine3( _TCHAR* argv[], const DWORD dwArg, Rational& rX, Rational& rY, Rational& rZ, bool& bSquareRoot )
   {
      if( 13 != dwArg )
      {
         if( 13 < dwArg )
         {
            wprintf( TEXT("Invalid parameters call. Too many parameters.\n\n ") );
         }
         else
         {
            wprintf( TEXT("Invalid parameters call. Too few parameters.\n\n ") );
         }

         return( false );
      }


      // Check that the sign of the number is OK.
      if( !string( argv[5] ).CompareNoCase( TEXT("-") ) && !string( argv[5] ).CompareNoCase( TEXT("+") ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the first rational number - sign is bad or missing.\n\n ") );

         return( false );
      }


      // Get the rX.
      MList< string > str( string( argv[6] ).Split( TEXT("/"), true, true ) );
      if( 2 != str.GetCount() )
      {
         wprintf( TEXT("Invalid parameters call: Error around the first rational number.\n\n ") );

         return( false );
      }

      DWORD dwN = str.GetHead()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwN ).CompareNoCase( *str.GetHead() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the first rational number.\n\n ") );

         return( false );
      }

      DWORD dwD = str.GetTail()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwD ).CompareNoCase( *str.GetTail() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the first rational number.\n\n ") );

         return( false );
      }

      rX = Rational( string( argv[5] ).CompareNoCase( TEXT("-" ) ), specialize< Integer as sX::Numerator >::Initialize( dwN ),
specialize< Integer as sX::Denominator >::Initialize( dwD ) );




      // Check that the sign of the functional is OK.
      if( !string( argv[7] ).CompareNoCase( TEXT("-") ) && !string( argv[7] ).CompareNoCase( TEXT("+") ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the second rational number - sign is bad or missing.\n\n ") );

         return( false );
      }

      // Check that rY is OK.
      str = string( argv[8] ).Split( TEXT("/"), true, true );
      if( 2 != str.GetCount() )
      {
         wprintf( TEXT("Invalid parameters call: Error around the second rational number.\n\n ") );
         return( false );
      }

      dwN = str.GetHead()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwN ).CompareNoCase( *str.GetHead() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the second rational number.\n\n ") );

         return( false );
      }

      dwD = str.GetTail()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwD ).CompareNoCase( *str.GetTail() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the second rational number.\n\n ") );

         return( false );
      }

      rY = Rational( string( argv[7] ).CompareNoCase( TEXT("-") ), specialize< Integer as sX::Numerator >::Initialize( dwN ),
specialize< Integer as sX::Denominator >::Initialize( dwD ) );



      // Check that the sign of the number inside the functional is OK.
      if( !string( argv[10] ).CompareNoCase( TEXT("-") ) && !string( argv[7] ).CompareNoCase( TEXT("+") ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the third rational number - sign is bad or missing.\n\n ") );

         return( false );
      }


      // Check that rZ is OK.
      str = string( argv[11] ).Split( TEXT("/"), true, true );
      if( 2 != str.GetCount() )
      {
         wprintf( TEXT("Invalid parameters call: Error around the third rational number.\n\n ") );

         return( false );
      }

      dwN = str.GetHead()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwN ).CompareNoCase( *str.GetHead() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the third rational number.\n\n ") );

         return( false );
      }

      dwD = str.GetTail()->operator unsigned long();
      if( !string( string::FF, TEXT("%u"), dwD ).CompareNoCase( *str.GetTail() ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the third rational number.\n\n ") );

         return( false );
      }

      rZ = Rational( string( argv[10] ).CompareNoCase( TEXT("-") ), specialize< Integer as sX::Numerator >::Initialize( dwN ),
specialize< Integer as sX::Denominator >::Initialize( dwD ) );



      // Get the function
      const string strFunction( argv[9] );

      if( !strFunction.Right( 1 ).CompareNoCase( TEXT("(") ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the function - missing opening bracket.\n\n ") );

         return( false );
      }

      if( !string( argv[12] ).CompareNoCase( TEXT(")") ) )
      {
         wprintf( TEXT("Invalid parameters call: Error around the function - missing closing bracket.\n\n ") );
         return( false );
      }

      // For now I have only square roots functor implementation.
      if( strFunction.LeftFromRight( 1 ).CompareNoCase( TEXT("sqrt") ) )
      {
         bSquareRoot = true;

         return( true );
      }

      wprintf( TEXT("Invalid parameters call: Error around the function - unknown functor name.\n\n ") );

      return( false );
   }
};


class Fibonacci : public DataSource
{
private:
   DWORD dwFn;
   DWORD dwFm;

private:
   void GetFibonacci( DWORD& dwFn, DWORD& dwFm )
   {
      const DWORD dwA = dwFn;
      dwFn = dwFm;
      dwFm = dwA + dwFm;
   }

public:
   Fibonacci() : dwFn( 0 ), dwFm( 1 ) {}
   ~Fibonacci() {}

   MEDIAN_TYPE GetNextNumber()
   {
      GetFibonacci( dwFn, dwFm );

      dwCurrentSequence++;

      return( MEDIAN_TYPE( Rational( specialize< Integer as sX::Numerator >::Initialize( dwFn ), specialize< Integer as
sX::Denominator >::Initialize( dwFm ) ) ) );
   }


   virtual bool ParseCommandLine( _TCHAR* argv[], const DWORD dwArg )
   {
      DWORD dwStartingIndex( 0 );
      if( !__super::ParseCommandLine2( argv, dwArg, dwStartingIndex, dwSequencesTotal ) )
      {
         return( false );
      }

      for( DWORD dwSkipIndex = 0; dwSkipIndex < dwStartingIndex; dwSkipIndex++ )
      {
         GetNextNumber();
      }

      dwCurrentSequence = 0;

      return( true );
   }
} g_Fibonacci;


class Binary : public DataSource
{
private:
   DWORD dwPower;

public:
   Binary() : dwPower( 0 ) {}
   ~Binary() {}

   MEDIAN_TYPE GetNextNumber()
   {
      dwPower++;
      dwCurrentSequence++;

      return( MEDIAN_TYPE( Rational( specialize< Integer as sX::Numerator >::Initialize( Integer( 2 ).ToPower( dwPower - 1 ) + 1 ),
specialize< Integer as sX::Denominator >::Initialize( Integer( 2 ).ToPower( dwPower ) ) ) ) );
   }

   virtual bool ParseCommandLine( _TCHAR* argv[], const DWORD dwArg )
   {
      DWORD dwStartingIndex( 0 );
      if( !__super::ParseCommandLine2( argv, dwArg, dwStartingIndex, dwSequencesTotal ) )
      {
         return( false );
      }

      for( DWORD dwSkipIndex = 0; dwSkipIndex < dwStartingIndex; dwSkipIndex++ )
      {
         GetNextNumber();
      }

      dwCurrentSequence = 0;

      return( true );
   }
} g_Binary;


class Equalidistant : public DataSource
{
private:
   MEDIAN_TYPE mtNext;
   MEDIAN_TYPE mtStep;

public:
   Equalidistant() : mtNext( Rational( specialize< Integer as sX::Numerator >::Initialize( 1 ), specialize< Integer as sX::Denominator
>::Initialize( 2 ) ) ),
                     mtStep( Rational( specialize< Integer as sX::Numerator >::Initialize( 1 ), specialize< Integer as sX::Denominator
>::Initialize( 2 ) ) )
   {
   }
   ~Equalidistant() {}

   MEDIAN_TYPE GetNextNumber()
   {
      mtNext = mtNext + mtStep;

      dwCurrentSequence++;

      return( mtNext );
   }


   // Method [0.5, 1] Equal-distant: E [start numerator] [end numerator] [denominator] [total sequences]  e.g. E 514 515 1000 10000
   virtual bool ParseCommandLine( _TCHAR* argv[], const DWORD dwArg )
   {
      Rational rStart;
      Rational rEnd;

      if( !__super::ParseCommandLine4( argv, dwArg, rStart, rEnd, dwSequencesTotal ) )
      {
         return( false );
      }

      if( 1 != dwSequencesTotal )
      {
         mtStep = (rEnd - rStart) / (dwSequencesTotal - 1);
         mtNext = rStart - mtStep;
      }
      else
      {
         mtStep = 0;
         mtNext = rStart;
      }

      return( true );
   }
} g_Equalidistant;
```

```
#ifdef QUADRATIC
class SingleQuadratic : public DataSource
{
private:
   MEDIAN_TYPE mtNext;

public:
   SingleQuadratic() : mtNext( Rational( true, specialize< Integer as sX::Numerator >::Initialize( 1 ), specialize< Integer as
sX::Denominator >::Initialize( 2 ) ), Functional< UniverseQuadraticFunctorType >( 1, UniverseQuadraticFunctorData ) )
   {
   }
   ~SingleQuadratic()
   {
   }

   MEDIAN_TYPE GetNextNumber()
   {
      dwCurrentSequence++;

      return( mtNext );
   }


   // Method a/b+c/d*sqrt(e/f) Quadratic: Q [numerator]/[denominator] [numerator]/[denominator] [numerator]/[denominator]  e.g. Q 1/2
3/4 5/8
   virtual bool ParseCommandLine( _TCHAR* argv[], const DWORD dwArg )
   {
      // rX + rY * sqrt( rZ )
      Rational rX;
      Rational rY;
      Rational rZ;
      bool bSquareRoot( false );

      MSmartPtr< Functor > ptrFunctor;

      if( !__super::ParseCommandLine3( argv, dwArg, rX, rY, rZ, bSquareRoot ) )
      {
         return( false );
      }


      if( bSquareRoot )
      {
         mtNext = Quadratic< UniverseQuadraticFunctorType >( rX, Functional< UniverseQuadraticFunctorType >( rY,
UniverseQuadraticFunctorType( rZ ) ) );
      }
      else
      {
         MASSERT( FALSE );
      }


      dwCurrentSequence = 0;
      dwSequencesTotal  = 1;

      return( true );
   }
} g_Quadratic;
#endif


void PrintTable( const bool bTrace,
                 const string& strFilename,
                 const string& strTitle,
                 MList< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > > listXn_ByValue,
                 MList< specialize< MEDIAN_TYPE, specialize< Xn, Clock > > > listXn_ByTime,
                 MList< specialize< MEDIAN_TYPE, specialize< Mn, Clock > > > listMedians )
{
   MStringEx< char >( MStringEx< char >::FF, "<html><head><title>%s</title></head><body style='font-family: Arial'>\r\n<table
border='0' cellpadding='3' cellspacing='0' >\r\n", strTitle.GetAsMultiByte().GetMemory() ).SaveAsFile( strFilename, false );


#ifdef QUADRATIC
   MEDIAN_TYPE mtMnSum( 0, Functional< UniverseQuadraticFunctorType >( 0, UniverseQuadraticFunctorData ) ), mtXnSum( 0, Functional<
UniverseQuadraticFunctorType >( 0, UniverseQuadraticFunctorData ) );
#else
   MEDIAN_TYPE mtMnSum, mtXnSum;
#endif

   if( bTrace ) { TRACE( TEXT("\n\n") ); }

   for( DWORD dwIndex = 0; dwIndex < listMedians.GetCount(); dwIndex++ )
   {
      mtMnSum += *listMedians[dwIndex];
      mtXnSum += *listXn_ByTime[dwIndex];

      if( 0 == dwIndex )
      {
         string( string::FF, TEXT("<tr>\
                                  <td style='font-size: 16px; padding-left:0px; text-align:center;'>Clocks</td>\
                                       <td style='font-size: 16px; padding-left:100px; text-align:center;'>Xn</td>\
                                       <td style='font-size: 16px; padding-left:100px; text-align:center;'>Xn Sorted</td>\
                                       <td style='font-size: 16px; padding-left:100px; text-align:center;'>Mn</td>\
                                       <td style='font-size: 16px; padding-left:100px; text-align:center;'>En</td>\
                                       <td style='font-size: 16px; padding-left:100px; text-align:center;'>M[n] - M[n-1]</td>\
                                       </tr>") ).GetAsMultiByte().AddToFile( strFilename, false );

         string( string::FF, TEXT("Clocks\tXn\tXn Sorted\tMn\tEn\tM[n] - M[n-1]\r\n") ).GetAsMultiByte().SaveAsFile( strFilename +
TEXT(".txt"), false );
      }

      const DWORD dwIndexBase( (DWORD)1 );
      string( string::FF, TEXT("<tr bgcolor='%s'>\
                                    <td style='text-align:right;'>%d</td>\
                                    <td style='text-align:right;'>%s</td>\
                                    <td style='text-align:right;'>%s</td>\
                                    <td style='text-align:right;'>%s</td>\
                                    <td style='text-align:right;'>%s</td>\
                                    <td style='text-align:right;'>%s</td>\
                                 </tr>"),
                                 0 == (1 & (dwIndexBase + dwIndex)) ? TEXT("#f4fff4") : TEXT("white"),
                          dwIndexBase + dwIndex,
                          (LPCTSTR)listXn_ByTime[dwIndex]->PrintAsHtmlCell(),
                          (LPCTSTR)listXn_ByValue[dwIndex]->GetLabel().PrintAsHtmlCell(),
                          (LPCTSTR)listMedians[dwIndex]->PrintAsHtmlCell(),
                          (LPCTSTR)(mtXnSum / (dwIndex + 1)).PrintAsHtmlCell(),
                          (LPCTSTR)(0 != dwIndex ? (*listMedians[dwIndex] - *listMedians[dwIndex-1]).PrintAsHtmlCell() : string(
TEXT("-") ) ) ).GetAsMultiByte().AddToFile( strFilename, false );

      if( bTrace )
      {
         TRACE( string( string::FF, TEXT("%.3d                %s                %s              %s            %s           %s\n"),
                     dwIndexBase + dwIndex,
                     (LPCTSTR)listXn_ByTime[dwIndex]->PrintAsDoubleInDecimalNotation( 16 ),
                     (LPCTSTR)listXn_ByValue[dwIndex]->GetLabel().PrintAsDoubleInDecimalNotation( 16 ),
                     (LPCTSTR)listMedians[dwIndex]->PrintAsDoubleInDecimalNotation( 16 ),
                     (LPCTSTR)(mtXnSum / (dwIndex + 1)).PrintAsDoubleInDecimalNotation( 16 ),
                     (LPCTSTR)(0 != dwIndex ? (*listMedians[dwIndex] - *listMedians[dwIndex-1]).PrintAsDoubleInDecimalNotation( 16 ) :
string( TEXT("-") ) ) ).GetAsMultiByte().GetMemory() );
      }

      string( string::FF, TEXT("%.3d\t%s\t%s\t%s\t%s\t%s\r\n"),
              dwIndexBase + dwIndex,
              (LPCTSTR)listXn_ByTime[dwIndex]->PrintAsDoubleInDecimalNotation( 16 ),
              (LPCTSTR)listXn_ByValue[dwIndex]->GetLabel().PrintAsDoubleInDecimalNotation( 16 ),
              (LPCTSTR)listMedians[dwIndex]->PrintAsDoubleInDecimalNotation( 16 ),
              (LPCTSTR)(mtXnSum / (dwIndex + 1)).PrintAsDoubleInDecimalNotation( 16 ),
              (LPCTSTR)(0 != dwIndex ? (*listMedians[dwIndex] - *listMedians[dwIndex-1]).PrintAsDoubleInDecimalNotation( 16 ) :
string( TEXT("-") ) )
            ).GetAsMultiByte().DecrementSize( MMemory< char >::MemUnits( 1 ) ).AddToFile( strFilename + TEXT(".txt") );
   }

   if( bTrace ) { TRACE( TEXT("\n\n") ); }

   MStringEx< char >( "</table>" ).AddToFile( strFilename, false );

   MStringEx< char >( "</body></html>\r\n" ).AddToFile( strFilename, false );
}
```

```
MEDIAN_TYPE Go( const DWORD dwDeepTrace, const string& strFilenameBase, const MEDIAN_TYPE mtStartPoint, MList< specialize< Pair<
MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > >& listXn_ByValue, MList< specialize< MEDIAN_TYPE, specialize< Xn, Clock > > >&
listXn_ByTime, MList< specialize< MEDIAN_TYPE, specialize< Mn, Clock > > >& listMedians )
{
   MASSERT( listXn_ByValue.IsEmpty() );
   MASSERT( listXn_ByTime.IsEmpty() );
   MASSERT( listMedians.IsEmpty() );


   listXn_ByValue.Empty();
   listXn_ByTime.Empty();
   listMedians.Empty();


   // Start the two Xn sequences: One sorted by values, and one sorted by clocks/time.
   listXn_ByValue.AddHead( new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >( specialize< Pair< MEDIAN_TYPE, DWORD
>, specialize< Xn, Value > >::Initialize( Pair< MEDIAN_TYPE, DWORD >( POINT_0, 1 ) ) ) );
   listXn_ByTime.AddHead(  new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >( specialize< MEDIAN_TYPE, specialize< Xn, Clock >
>::Initialize( 0 ) ) );

   listXn_ByValue.AddTail( new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >( specialize< Pair< MEDIAN_TYPE, DWORD
>, specialize< Xn, Value > >::Initialize( Pair< MEDIAN_TYPE, DWORD >( mtStartPoint, 2 ) ) ) );
   listXn_ByTime.AddTail(  new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >( specialize< MEDIAN_TYPE, specialize< Xn, Clock >
>::Initialize( mtStartPoint ) ) );

   listXn_ByValue.AddTail( new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >( specialize< Pair< MEDIAN_TYPE, DWORD
>, specialize< Xn, Value > >::Initialize( Pair< MEDIAN_TYPE, DWORD >( POINT_1, 3 ) ) ) );
   listXn_ByTime.AddTail(  new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >( specialize< MEDIAN_TYPE, specialize< Xn, Clock >
>::Initialize( 1 ) ) );

   listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >( specialize< MEDIAN_TYPE, specialize< Mn, Clock >
>::Initialize( listXn_ByValue[0]->GetLabel() ) ) );
   if( !strFilenameBase.IsEmpty() ) PrintTable( false, strFilenameBase + string( string::FF, TEXT("%.5d"), + listMedians.GetCount() )
+ TEXT(".htm"), TEXT(""), listXn_ByValue, listXn_ByTime, listMedians );

   listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >( specialize< MEDIAN_TYPE, specialize< Mn, Clock >
>::Initialize( (listXn_ByValue[0]->GetLabel() + listXn_ByValue[1]->GetLabel()) / 2 ) ) );
   if( !strFilenameBase.IsEmpty() ) PrintTable( false, strFilenameBase + string( string::FF, TEXT("%.5d"), + listMedians.GetCount() )
+ TEXT(".htm"), TEXT(""), listXn_ByValue, listXn_ByTime, listMedians );

   listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >( specialize< MEDIAN_TYPE, specialize< Mn, Clock >
>::Initialize( listXn_ByValue[1]->GetLabel() ) ) );
   if( !strFilenameBase.IsEmpty() ) PrintTable( false, strFilenameBase + string( string::FF, TEXT("%.5d"), + listMedians.GetCount() )
+ TEXT(".htm"), TEXT(""), listXn_ByValue, listXn_ByTime, listMedians );


   MEDIAN_TYPE mtSum_n( listXn_ByValue[0]->GetLabel() + listXn_ByValue[1]->GetLabel() + listXn_ByValue[2]->GetLabel() );


   MList< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > >::MP< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize<
Xn, Value > > >* pMedian( listXn_ByValue.GetHeadMP()->GetNext() );

   MEDIAN_TYPE mtMed_n( pMedian->GetObject()->GetLabelRef() );

   DWORD dwCurrentTrace = 1;
   for( MEDIAN_TYPE mtX_n_plus_1 = (listXn_ByValue.GetCount() + 1) * mtMed_n - mtSum_n; mtX_n_plus_1 != mtMed_n; mtX_n_plus_1 =
(listXn_ByValue.GetCount() + 1) * mtMed_n - mtSum_n )
   {
      // Lemma 12. X[n+1] >= M[n].
      MASSERT( mtX_n_plus_1 >= mtMed_n );


      mtSum_n += mtX_n_plus_1;


      // Add the new Xn in the clock sorted sequence.
      listXn_ByTime.AddTail( new specialize< MEDIAN_TYPE, specialize< Xn, Clock > >( specialize< MEDIAN_TYPE, specialize< Xn, Clock >
>::Initialize( mtX_n_plus_1 ) ) );


      // Insert dX_np1 in the sequence and find the new Median.h
      for( MList< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > >::MP< specialize< Pair< MEDIAN_TYPE, DWORD >,
specialize< Xn, Value > > >* pMP( pMedian ); NULL != pMP; pMP= pMP->GetNext() )
      {
         if( mtX_n_plus_1 < pMP->GetObject()->GetLabel() )
         {
            // Insert median here. It is important that the inequality is strict! <= would insert the
            // element in the correct place in respect to the values, but that will make the pointer to
            // the median to point to the wrong place as the sequence will be shifted with one element.
            pMP = listXn_ByValue.AddBehind( pMP->GetPrevious(), new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >(
specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >::Initialize( Pair< MEDIAN_TYPE, DWORD >( mtX_n_plus_1,
listXn_ByTime.GetCount() ) ) ) );

            break;
         }
      }

      if( listXn_ByTime.GetCount() != listXn_ByValue.GetCount() )
      {
         // The mtX_n_plus_1 has not yet been added.
         listXn_ByValue.AddTail( new specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > >( specialize< Pair< MEDIAN_TYPE,
DWORD >, specialize< Xn, Value > >::Initialize( Pair< MEDIAN_TYPE, DWORD >( mtX_n_plus_1, listXn_ByTime.GetCount() ) ) ) );
      }


      if( 0 == (listXn_ByValue.GetCount() & 1) )
      {
         mtMed_n = (pMedian->GetObject()->GetLabelRef() + pMedian->GetNext()->GetObject()->GetLabelRef())/2;
         pMedian = pMedian->GetNext();
      }
      else
      {
         mtMed_n = pMedian->GetObject()->GetLabelRef();
      }


      listMedians.AddTail( new specialize< MEDIAN_TYPE, specialize< Mn, Clock > >( specialize< MEDIAN_TYPE, specialize< Mn, Clock >
>::Initialize( mtMed_n ) ) );


      if( !strFilenameBase.IsEmpty() )
      {
         if( dwDeepTrace == dwCurrentTrace )
         {
            PrintTable( false, strFilenameBase + string( string::FF, TEXT("%.6d.htm"), + listMedians.GetCount() ), TEXT(""),
listXn_ByValue, listXn_ByTime, listMedians );

            dwCurrentTrace = 0;
         }

         wprintf( TEXT("Medians Count: %d\n "), listMedians.GetCount() );
      }

      dwCurrentTrace++;
   }


   MASSERT( listMedians.GetCount() == listXn_ByValue.GetCount() );

   return( mtMed_n );
}


struct CalculationData
{
   const DWORD       dwIndex;
   const MEDIAN_TYPE mtStartPoint;
   const string      strDataFolder;
   const HANDLE      heFinished;
   const DWORD        dwDeepTrace;

   CalculationData( const DWORD dwIndex,
                    const MEDIAN_TYPE& mtStartPoint,
                    const string& strDataFolder,
                    const HANDLE heFinished,
                    const DWORD dwDeepTrace )
      :  dwIndex( dwIndex ),
         mtStartPoint( mtStartPoint ),
         strDataFolder( strDataFolder ),
         heFinished( heFinished ),
         dwDeepTrace( dwDeepTrace )
   {
   }
   ~CalculationData()
   {
   }
};


DWORD WINAPI CalculateHalt( LPVOID pvCalculationData )
{
   MSmartPtr< CalculationData > prtData( (CalculationData*)pvCalculationData );


   // Prepare recording location and properties.
   const string2z strFolder( prtData->strDataFolder + string( string::FF, TEXT("\\%.10d"), (LPCTSTR)prtData->dwIndex ) );
   const string   strRecord( string( strFolder.GetMemory() ) + TEXT("\\Record_") );


   SHFILEOPSTRUCT sfs = { 0 };
   sfs.wFunc  = FO_DELETE;
   sfs.pFrom  = strFolder.GetMemory();
   sfs.fFlags = FOF_ALLOWUNDO | FOF_SILENT | FOF_NOCONFIRMATION | FOF_NOERRORUI | FOF_NOCONFIRMMKDIR;

   ::SHFileOperation( &sfs );
   ::CreateDirectory( strFolder, NULL );


   // Calculation data holders.
   MList< specialize< Pair< MEDIAN_TYPE, DWORD >, specialize< Xn, Value > > > listXn_ByValue;
   MList< specialize< MEDIAN_TYPE, specialize< Xn, Clock > > > listXn_ByTime;
   MList< specialize< MEDIAN_TYPE, specialize< Mn, Clock > > > listMedians;


   // Do the calculation.
   const MEDIAN_TYPE mtHalt( Go( prtData->dwDeepTrace, prtData->dwDeepTrace ? strRecord : TEXT(""), prtData->mtStartPoint,
listXn_ByValue, listXn_ByTime, listMedians ) );


   // Print the final data.
   PrintTable( false, strRecord + TEXT("Final.html"), TEXT(""), listXn_ByValue, listXn_ByTime, listMedians );


   // Find the Centre.
   Pair< DWORD, DWORD > prCentre;
   for( DWORD dwIndex = 0; dwIndex < listXn_ByValue.GetCount(); dwIndex++ )
   {
      DWORD dwPosS( 0 );
      DWORD dwPosF( 0 );

      if( listXn_ByValue[dwIndex]->GetLabelRef() == mtHalt )
      {
         dwPosS = dwIndex;
         dwPosF = dwIndex;

         for( DWORD dwIndexInner = dwIndex + 1; (dwIndexInner < listXn_ByValue.GetCount()) && (listXn_ByValue[dwIndex]->GetLabelRef()
== listXn_ByValue[dwIndexInner]->GetLabelRef()); dwIndexInner++ )
         {
            dwIndex = dwPosF = dwIndexInner;
         }

         prCentre = Pair< DWORD, DWORD >( dwPosS, dwPosF - dwPosS + 1 );

         break;
      }
   }


   // Find Duplicates
   MList< Pair< MEDIAN_TYPE, Pair< DWORD, DWORD > > > listDuplicate;
   for( DWORD dwIndex = 0; dwIndex < listXn_ByValue.GetCount() - 1; dwIndex++ )
   {
      DWORD dwPosS( 0 );
      DWORD dwPosF( 0 );

      if( listXn_ByValue[dwIndex]->GetLabelRef() == listXn_ByValue[dwIndex+1]->GetLabelRef() )
      {
         dwPosS = dwIndex;
         dwPosF = dwIndex;
         for( DWORD dwIndexInner = dwIndex + 1; (dwIndexInner < listXn_ByValue.GetCount()) && (listXn_ByValue[dwIndex]->GetLabelRef()
== listXn_ByValue[dwIndexInner]->GetLabelRef()); dwIndexInner++ )
         {
            dwIndex = dwPosF = dwIndexInner;
         }
      }

      if( dwPosS != dwPosF )
      {
         listDuplicate.AddTail( new Pair< MEDIAN_TYPE, Pair< DWORD, DWORD > >( listXn_ByValue[dwPosS]->GetLabelRef(), Pair< DWORD,
DWORD >( dwPosS, dwPosF - dwPosS + 1 ) ) );
      }
   }


   const string strMatchesFilename( 1 < listDuplicate.GetCount() ? strRecord + TEXT("Kernels.txt") : TEXT("-") );

   // Record this value data.
   string( string::FF,
           TEXT("%d\t%s\t\"%s\"\t\t\t%s\t\"%s\"\t\t\t%s\t\"%s\"\t\t\t%d\t%d\t%d\t\"%s\"\r\n"),
           prtData->dwIndex,                                                                           // index
           prtData->mtStartPoint.PrintAsDoubleInDecimalNotation( 16 ).GetMemory(),                     // start number
           prtData->mtStartPoint.Print().GetMemory(),                                                  // start number as rational
           mtHalt.PrintAsDoubleInDecimalNotation( 16 ).GetMemory(),                                    // halt value
           mtHalt.Print().GetMemory(),                                                                 // halt value as rational
           listXn_ByValue.GetTail()->GetLabelRef().PrintAsDoubleInDecimalNotation( 16 ).GetMemory(),   // Max Value
           listXn_ByValue.GetTail()->GetLabelRef().Print().GetMemory(),                                // Max Value as rational
           prCentre.GetLabel(),                                                                        // centre position
           prCentre.GetValue(),                                                                        // centre size
           listXn_ByValue.GetCount(),                                                                  // transit clocks
           strMatchesFilename.GetMemory()                                                              // Kernels
           ).GetAsMultiByte().AddToFile( prtData->strDataFolder + DATA_FILE_NAME, false );



   // Record the Kernels
   if( 1 < listDuplicate.GetCount() )
   {
      // Overwrite any existing file and set file header.
      MStringEx< char >( "Value\tPosition\tRepeats\r\n" ).SaveAsFile( strMatchesFilename, false );

      for( DWORD dwIndex = 0; dwIndex < listDuplicate.GetCount(); dwIndex++ )
      {
         MStringEx< char >( MStringEx< char >::FF,
                            "%s\t%d\t%d\r\n",
                            listDuplicate[dwIndex]->GetLabelRef().PrintAsDoubleInDecimalNotation( 16 ).GetAsMultiByte().GetMemory(),
                            listDuplicate[dwIndex]->GetValueRef().GetLabel(),
                            listDuplicate[dwIndex]->GetValueRef().GetValueRef() ).AddToFile( strMatchesFilename, false );
      }
   }


   SetEvent( prtData->heFinished );


   return( 0 );
}


void PrintProgramCall()
{
   wprintf( TEXT("Parameters:\n ") );
   wprintf( TEXT("Target path: must exists                         e.g. C:\\path\n ") );
   wprintf( TEXT("Max number of threads: 1 to %d                   e.g. 12\n "), MAX_COMPUTATION_THREAD_COUNT );
   wprintf( TEXT("Record every X-th intermediate sequence: 0 - record none or integer number \n ") );
#ifndef QUADRATIC
   wprintf( TEXT("Method Fibonacci #:            F [start index] [total sequences]  e.g. F 41 25\n ") );
   wprintf( TEXT("Method (2^(i-1)+1)/2^i:        B [start index] [total sequences]  e.g. B 41 25\n ") );
   wprintf( TEXT("Method [0.5, 1] Equal-distant: E [start numerator] [end numerator] [denominator] [total sequences]  e.g. E 514 515
1000 10000\n ") );
   wprintf( TEXT("\n\n" ) );
   wprintf( TEXT("Example call: app-name C:\\folder\\fib 12 0 F 41 25\n") );
   wprintf( TEXT("Example call: app-name C:\\folder\\fib 12 0 E 514 515 1000 10000") );
#else
   wprintf( TEXT("Method a/b+c/d*sqrt(e/f) Quadratic: [+ or -] Q [numerator]/[denominator] [+ or -] [numerator]/[denominator]
sqrt([numerator]/[denominator])  e.g. Q 1/2 3/4 5/8\n ") );
   wprintf( TEXT("\n\n" ) );
   wprintf( TEXT("Example call: app-name C:\\m1\\fib 1 Y Q - 1/2 + 1/2 sqrt( + 5/1 )") );
```

```
#endif
   wprintf( TEXT("\n\n" ) );
}


int _tmain(int argc, _TCHAR* argv[])
{
   wprintf( TEXT("\nMean-Median problem computation application\n") );
   wprintf( TEXT("Copyright (c) 2010 - 2011 Miroslav Bonchev Bonchev. All rights reserved.\n\n" ) );
   wprintf( TEXT("This software uses the Atomic Memory Model by Miroslav Bonchev Bonchev.\n" ) );
   wprintf( TEXT("This software uses the Object Specialization Model by Miroslav Bonchev Bonchev.\n" ) );
   wprintf( TEXT("This software uses the Proper Numbers Library by Miroslav Bonchev Bonchev.\n" ) );
   wprintf( TEXT("\n\nFor more information please visit:\n\n" ) );
   wprintf( TEXT("http://www.mbbsoftware.com\n" ) );
   wprintf( TEXT("http://www.mbbsoftware.com/Software/AtomicMemoryModel/Default.aspx\n" ) );
   wprintf( TEXT("http://www.mbbsoftware.com/Software/ProperNumbersLibrary/Default.aspx\n\n" ) );


   if( 7 > argc )
   {
      wprintf( TEXT("Invalid parameters call. Too few parameters.\n\n ") );

      PrintProgramCall();

      return( 0 );
   }


   // Retrieve the target path.
   const string strTargetPath( argv[1] );
   const DWORD dwPathAttrib( ::GetFileAttributes( strTargetPath ) );

   if( INVALID_FILE_ATTRIBUTES == dwPathAttrib )
   {
      wprintf( TEXT("Invalid parameters call: Path does not exists.\n\n ") );

      PrintProgramCall();

      return( 0 );
   }

   if( 0 == (FILE_ATTRIBUTE_DIRECTORY & dwPathAttrib) )
   {
      wprintf( TEXT("Invalid parameters call: The entered target path exists but is a file.\n\n ") );

      PrintProgramCall();

      return( 0 );
   }


   // Retrieve the computational threads count.
   const DWORD dwComputationalThreads( string( argv[2] ).operator unsigned long() );
   if( !IsInRange< DWORD, true, DWORD, true, DWORD >( 1, dwComputationalThreads, MAX_COMPUTATION_THREAD_COUNT ) )
   {
      wprintf( TEXT("Invalid parameters call: The number of computational threads must be between 1 and 24 inclusive.\n\n ") );

      PrintProgramCall();

      return( 0 );
   }


   // Retrieve the trace depth.
   const DWORD dwDeepTrace( string( argv[3] ).operator unsigned long() );
   if( !string( string::FF, TEXT("%u"), dwDeepTrace ).CompareNoCase( argv[3] ) )
   {
      wprintf( TEXT("Invalid parameters call: The deep trace parameter is invalid. Please enter a decimal number from 0 to 2^32-1.\n\n
") );

      PrintProgramCall();

      return( 0 );
   }


   // Retrieve the required computation method.
   DataSource* pDataSource( NULL );
   // Fibonacci Data Source
   if( string( argv[4] ).CompareNoCase( TEXT("F") ) )
   {
      pDataSource = &g_Fibonacci;
   }

   // Binary Data Source
   if( string( argv[4] ).CompareNoCase( TEXT("B") ) )
   {
      pDataSource = &g_Binary;
   }

   // Equal-distant Data Source
   if( string( argv[4] ).CompareNoCase( TEXT("E") ) )
   {
      pDataSource = &g_Equalidistant;
   }

#ifdef QUADRATIC
   // Quadratic Data Source
   if( string( argv[4] ).CompareNoCase( TEXT("Q") ) )
   {
      pDataSource = &g_Quadratic;
   }
#endif

   // Check that data source was specified.
   if( NULL == pDataSource )
   {
      wprintf( TEXT("Invalid parameters call: No valid data source was specified. Please enter a valid data source data generator
identification letter.\n\n ") );

      PrintProgramCall();

      return( 0 );
   }


   // Go to the required location in the phase space, from where the user wants to start.
   if( !pDataSource->ParseCommandLine( argv, argc ) )
   {
      PrintProgramCall();

      return( 0 );
   }


   // Now, get to work.
   HANDLE heEventStop[MAX_COMPUTATION_THREAD_COUNT];
   Dictionary< HANDLE, MHandle > dictThreadHandle;


   // Create the thread completion events.
   for( DWORD dwIndex = 0; dwIndex < MMIN< DWORD, DWORD, DWORD >( dwComputationalThreads, pDataSource->GetRemainingSequencesCount() );
dwIndex++ )
   {
      dictThreadHandle.Add( heEventStop[dwIndex] = CreateEvent( NULL, TRUE, TRUE, NULL ), NULL );
   };


   wprintf( TEXT("\nLaunching threads:\n") );
   MStringEx< char >( "Mean-Median problem computation application\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "Copyright (c) 2010 - 2011 Miroslav Bonchev Bonchev - http://www.MBBSoftware.com. All rights reserved.\r\n\r\n"
).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "This software uses the Atomic Memory Model by Miroslav Bonchev Bonchev -
http://www.mbbsoftware.com/Software/AtomicMemoryModel/Default.aspx\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "This software uses the Object Specialization Model by Miroslav Bonchev Bonchev -
http://www.mbbsoftware.com\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "This software uses the Proper Numbers Library by Miroslav Bonchev Bonchev -
http://www.mbbsoftware.com/Software/ProperNumbersLibrary/Default.aspx\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( MStringEx< char >::FF, "Program started as: %s\r\n", ::GetCommandLineA() ).AddToFile( strTargetPath +
DATA_FILE_NAME, false );
   MStringEx< char >( "\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >(
"=====================================================================================================================================
=================================================\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
```

```
   MStringEx< char >( "[index startF startS haltF haltS maxF maxS centrePos centreSize transitClocks Kernels] =
textread('C:\\mm\\fib\\Data.txt','%d %f %s %f %s %f %s %d %d %d %s', 'headerlines', 14 );\r\n" ).AddToFile( strTargetPath +
DATA_FILE_NAME, false );
   MStringEx< char >(
"=====================================================================================================================================
=================================================\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "Index\tStart Number\tStart Number as Rational\t\t\tHalt Value\tHalt Value as Rational\t\t\tMax Value\tMax Value
as Rational\t\t\tCentre Position\tCentre Size\tTransit Clocks\tKernels\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );
   MStringEx< char >( "---------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------\r\n" ).AddToFile( strTargetPath + DATA_FILE_NAME, false );


   DWORD dwSequenceIndex( 0 );

   while( !dictThreadHandle.IsEmpty() )
   {
      const DWORD dwSignalledIndex( WaitForMultipleObjects( dictThreadHandle.GetCount(), heEventStop, FALSE, 1000 ) );

      if( IsInRange< DWORD, true, DWORD, true, DWORD >( WAIT_OBJECT_0, dwSignalledIndex, WAIT_OBJECT_0 + dictThreadHandle.GetCount() -
1 ) )
      {
         const DWORD dwEventIndex( dwSignalledIndex - WAIT_OBJECT_0 );

         if( !pDataSource->HasMoreToDo() )
         {
            // There is no more data. Just wait for all threads to complete.
            // Remove the event for the array.
            dictThreadHandle.Remove( heEventStop[dwEventIndex] );

            // Re-establish the events array excluding the destroyed event.
            for( DWORD dwEvent = 0; dwEvent < dictThreadHandle.GetCount(); dwEvent++ )
            {
               heEventStop[dwEvent] = dictThreadHandle.GetByIndex( dwEvent ).GetLabel();
            };

            wprintf( TEXT("Thread [%.2d] finished, %d threads remain to finish.\n"), dictThreadHandle.GetCount(),
dictThreadHandle.GetCount() );

            continue;
         }


         ResetEvent( heEventStop[dwEventIndex] );

         // Get the next number to compute.
         const MEDIAN_TYPE mtNumber( pDataSource->GetNextNumber() );

         // Launch the computational thread.
         dictThreadHandle[heEventStop[dwEventIndex]] = CreateThread( NULL, 0, CalculateHalt, (LPVOID)new CalculationData(
dwSequenceIndex, mtNumber, strTargetPath, heEventStop[dwEventIndex], dwDeepTrace ), 0, NULL );

         // Set lower thread priority - make the system usable for other purposes while computing.
         SetThreadPriority( dictThreadHandle[heEventStop[dwEventIndex]], THREAD_PRIORITY_IDLE );

         // Display the thread starting information.
         SYSTEMTIME lt;
         ::GetLocalTime( &lt );

         wprintf( TEXT("%.4d-%.2d-%.2d %.2d:%.2d:%.2d Thread[%.2d] = [%.3d] %s\n"), lt.wYear, lt.wMonth, lt.wDay, lt.wHour,
lt.wMinute, lt.wSecond, dwEventIndex, dwSequenceIndex, (LPCTSTR)mtNumber.Print() );

         dwSequenceIndex++;
      }
   }


   wprintf( TEXT("Job done!\n") );


   return( 1 );
}
```

4.4. Matlab Resources

The Mean Medain Map application records the data for each halted sequences in a file with predefined name #define DATA_FILE_NAME TEXT("\\Data.txt") in folder supplied to the application through a command line switch. The recorded data is in text format suitable for importing from data processing and visualizing application such as MATLAB. The script below is a MATLAB script importing the data from the data file produced by the Mean Median Map application referred to by the DataFileName variable. Since the data file is produced by concurrent threads the order in which they complete and record their information is generally not the same as the order in which they were spawned. For this reason the script below sorts the data by sequence-index before plotting it. The other rows sort commands are to arrange the data in ways suitable for different plots which one may desire to plot. Note that although the Mean Median Map uses arbitrary precision Rational and Quadratic numbers the data file is recorded using double precision (64 bit) floating point numbers so external data processing applications could read the data. This of course reduces the precision of the recorded information. The human interpretation html files however produced with each completed sequence or requested deep log contain the Rational/Quadratic numbers together with their double representation.

Listing 10. MATLAB script for importing and visualising data from multiple sequences.

```
clear;

DataFileName = 'C:\mm\equ_514\Data.txt';

format long;

[index startF startS haltF haltS maxF maxS centrePos centreSize transitClocks Kernels] = textread( DataFileName,'%d␣%f␣%s␣%f␣%s␣%f
      ␣%s␣%d␣%d␣%d␣%s', 'headerlines', 14 );

data  = [index startF haltF maxF centrePos centreSize transitClocks];

indexIndex = 1;
indexStart = 2;
indexHalt  = 3;
indexMax   = 4;
indexCP    = 5;
indexCS    = 6;
indexClock = 7;

% Sorted by Index/Start Value.
dataIndex = sortrows( data, indexIndex );
dataStart = sortrows( data, indexStart );
dataHalt  = sortrows( data, indexHalt );
dataMax   = sortrows( data, indexMax );
dataCP    = sortrows( data, indexCP );
dataCS    = sortrows( data, indexCS );
dataClock = sortrows( data, indexClock );

subplot( 2, 1, 1 );
plot( dataIndex( :, indexIndex ), dataIndex( :, indexHalt ), 'r', dataIndex( :, indexIndex ), dataIndex( :, indexMax ), 'b' );
set( legend( 'Index␣vs␣Halt␣Value', 'Index␣vs␣Max␣Value' ),'Interpreter','none' )

subplot( 2, 1, 2 );
plot( dataIndex( :, indexIndex ), dataIndex( :, indexClock ), 'b' );
set( legend( 'Index␣vs␣Iterations␣Count␣Value' ),'Interpreter','none' )
```

Before terminating work threads record the final X_n, X_n^{Sort}, M_n, E_n and $M_n{-}M_{n-1}$ sequences in Record_Final.html and Record_Final.html.txt files in dedicated for each sequence subfolder of the path passed to the Mean Median Map application at start up. The first file is for human interpretation while the text file is designed for importing by data processing and visualizing application, such as MATLAB. If deep logging is requested via the dedicated command line switch the Mean Median Map application will log the same sequences periodically as they develop in accordance with the requested recording frequency. The data in the machine interpretation file is recorded using 64 bit floating point numbers degrading the arbitrary precision to double which determines precision errors due the nature and the limited size of the double floating point numbers. Further the Quadratic class in particular may incur large conversion errors once its true numbers become too large. Some additional work on the Quadratic class to improve the conversion form Quadratic to floating point might be useful. The script below imports sequence data and displays some useful plots from it.

Listing 10. MATLAB script for importing and visualising data of a Mean Median Map sequence.

```
clear;

DataFileName = 'C:\mm\equ_514\0000000181\Record_Final.html.txt';

format long;
[index Xn_time Xn_sort Mn En Delta] = textread( DataFileName,'%d␣%f␣%f␣%f␣%f␣%f', 'headerlines', 2 );

data  = [index Xn_time Xn_sort Mn En Delta];

plot( index, Xn_time, 'g', index, Xn_sort, 'r', index, Mn, 'b' );
set( legend( 'Xn', 'Xn-Sort', 'Mn' ), 'Interpreter', 'none' );
title( sprintf( 'Starting␣Set␣(0,␣%f,␣1)␣-␣Halt␣Value␣=␣%f,␣Iteration␣Count␣=␣%d', Xn_time( 1 ), Mn( size( Mn, 1 ) ), size( Mn, 1
      ) - 2 ) );
```

4.5. Other Classes and Resources

There are other generic resources required to compile the Mean Median Map application published in the next section. These facilities are generic linked list, pair, ASSERTs and other similar. Although not all files listed below are necessary for the Mean Median Map there are cross references between them and thus it is advisable to download and use them all. The classes and other content in the files is a copyrighted property of the author and are published under the MIT Open Source Software License Agreement as advised on the top of each file. The files listed below can be downloaded from `http://www.MBBSoftware.com/Software/ProperNumbersLibrary/Default.aspx`.

1. Common.h - This file provides some common definitions, such as ASSERT and TRACE.
2. Dictionary.h - This file contains a (simple) dictionary class.
3. Integer.h - This file contains the Integer class.
4. MAtom.h - This file contains a Example Implementation of the Atomic Memory Model Phase Two.
5. MHandle.h - This file contains a system HANDLE wrapper class.
6. MList.h - This file contains a generic list class.
7. MMemory.h - This file contains a Example Implementation of the Atomic Memory Model Phase One.
8. MSmartPtr.h - This file contains a smart pointer implementation class.
9. Pair.h - This file contains a generic pair class.
10. Rational.h - This file contains the Rational class.
11. StringEx.h - This file contains a generic string class.
12. WRID.h - This file contains a Windows Resource Identifier wrapper class.

Part V

Conclusion

Although there is no formalized proof for the main conjecture this project explained the work of the Mean Median Map. The weak conjecture was proved and by all the known facts this project supports the main conjecture.

This paper also presents the Object Specialization Model, which although not directly related to the Mean Median Map was extensively used in the software modelling it. The Object Specialization Model helped the rapid development of the software and had to be included in this paper as it is not previously known. The required background work on the Mean Median Map produced a new library for well-defined Rational and Integer numbers, which also utilized the Object Specialization Model in order to achieve the highest known standard.

There are certainly more things that require further attention. On the Mean Median Map the main conjecture proof need to be formalized. Further the continuity conjecture has not been investigated in this project. On the software part it is most interesting to resolve problem 27. A more trivial development is adding more mathematical function to the different numeric classes, such as $sin(x)$, $cos(x)$, $log(x)$, $exp(x)$, hyperbolic functions, etc, as well as improve the interfacing (print, read).

In closing, the author would like thank all who have in one way or another halped for producing this work and stress once again that the reason to understand the map and its operation map was the cooperation and equal treat of Computer Science and Mathematics. This includes but is not limited to the enforced by Computer Science constructivism.

References

[1] Chamberland, M. and Martelli, M., 2007, Journal of Difference Equations and Applications

[2] Schultz, H. and Shiflett, R., 2005, M7m sequences. The College Mathematics Journal, **36**, 191-198