CO620

Akiko Kathrin Uriu

Arbitrary Precision in Logic Programming Using Continued Fractions

Akiko Kathrin Uriu
Computing Laboratory, University of Kent
kau2@kent.ac.uk

Abstract

When a number is represented as a continued fraction, then it comes with a natural error bound. Continued fractions can be expressed as digit streams. Arbitrary precision can be achieved by truncating the stream appropriately. Introducing more terms will refine the representation whilst preserving the ability for further refinement. The value of continued fraction arithmetic has been recognized by the functional programming community, because continued fractions can be naturally implemented as lazy streams, but is not as widely known in logic programming. Delay declarations can be used to orchestrate the control needed to compute numeric results lazily to the required degree of precision. Irrational numbers can be represented by infinite continued fractions, which, if they have recurring patterns, can be represented exactly by rational trees. This project demonstrates how continued fraction arithmetic works and how it can be implemented using logic programming features to achieve the desired precision of a result.

1. Introduction

One fundamental idea behind constraint logic programming is that the computational domain is a parameter that is instantiated with a particular constraint solver. Constraint programming systems, notably BNR(R) Prolog [28], Numerica [35] and more recently the ILOG Solver [17], provide solvers that support interval arithmetic [24]. The unique feature of these solvers is that they provide a way of bounding numerical errors. If the interval of the result is too coarse, then the computation is repeated until desired precision is obtained. However it is a mistake to think that interval arithmetic is the only way to bound errors with intervals. If a number is represented as a continued fraction, then the continued fraction comes with a natural error bound. Consider, for example, the continued fraction

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}$$

This represents an approximation the Golden Ratio that is $\frac{1+\sqrt{5}}{2}$. The continued fraction can be represented as the open ended stream (list) `[1,1,1 | Rest]`. Instantiating Rest to an open ended stream, for example, assigning `Rest=[1,1,1 | NewRest]` and thereby introducing three more terms, will refine the representation whilst preserving the ability for further refinement. One remarkable property of continued fractions is that, when the stream is truncated to the nth and $(n + 1)^{th}$ terms, the resulting numbers define the endpoints of an interval in which the continued fraction resides. Furthermore, these truncations, formally known as the partial quotients, converge to the continued fraction itself and therefore any precision can be obtained by making n suitably large. In this example the first ten partial quotients, or convergents, obtained by evaluating the continued fraction to successively deeper terms are $\frac{1}{1}$, $\frac{2}{1}$, $\frac{3}{2}$, $\frac{5}{3}$, $\frac{8}{5}$, $\frac{13}{8}$, $\frac{21}{13}$, $\frac{34}{21}$, $\frac{55}{34}$, and $\frac{89}{55}$. Note the oscillatory nature of these partial quotients.

The deliberate study of continued fractions dates back to the 17th century text *Arithmetica Infinitorium*. However, although the value of continued fractions as a representation was long recognised, continued fraction arithmetic is notoriously difficult and the field has been somewhat neglected. In fact, it was Khinchin who wrote in the authoritative book Continued Fractions

[18], that: “even the problem of finding the continued fraction for a sum from the continued fraction representing the addends is exceedingly complicated, and unworkable in computational practice”. Ironically, the breakthrough came in two unpublished works, both by Gosper, who presented tractable algorithms for continued fraction arithmetic [1], [12], and remarked “that continued fractions are not only perfectly amenable to arithmetic, they are amenable to perfect arithmetic”. The definitive reference on continued fraction arithmetic is now taken to be Vuillemin [38], [39], thought this paper assumes a high degree of mathematical expertise. (Knuth [20] devotes little space to continued fraction arithmetic. He gives an exercise of computing 3x, where x is a continued fraction, and rates this problem as being somewhere between “Medium” and “Moderately hard”.) The value of continued fraction arithmetic has been recognised by a number of researchers, mostly in functional programming, because continued fractions can be naturally implemented as lazy streams [22], [33], [26], [21], [31]. Although streaming techniques have been applied in functional programming [11], they are not as widely known in logic programming. One example of implementing lazy streams in logic programming is given by Gregory, who proposed a solution to the Hamming problem to “generate in increasing order the sequence of all numbers divisible by no primes other than 2, 3 or 5” [14]. One of Gregory’s crucial insights was that delay declarations can be used to realise lazy streams. Our work builds on this fundamental idea by showing how delay declarations can control continued fraction arithmetic and enforce the required level of precision. Another technique we draw on is rational trees. Rational trees arise when the occur-check is relaxed. A rational tree is a tree that is augmented with back-arcs. Consider the tree `f(a,g)`, with root `f` and with child nodes `a` and `g`. A rational tree arises when a back-arc is added between `g` and `f`. This tree, though finitely representable, contains the infinite subtree `g(f(a, g(f(a,g···))))`. Rational trees provide a valuable way to implement streams that contain repeating subsequences. This paper draws together a number of diverse threads in the literature. And therefore, for clarity, we list our contributions as follows:

careful application of language features provided by second-generation logic programming systems can be used to realise arbitrary precision arithmetic;

delay declarations can be used to orchestrate the control needed to compute numeric results lazily to the required degree of precision;

continued fractions with recurring patterns can be represented exactly by rational trees;

this paper synthesises the continued fraction literature from a logic programming perspective,

and it corrects some errors that have crept into a tutorial [36].

This project shows that language features of second generation systems, specifically delay declarations and rational trees, can be used to support arbitrary precision arithmetic.

2. Background

Arbitrary precision, continued fractions, rational trees and delay declarations may be concepts that are new to the reader and sound complicated and difficult. Although some of the literature on the subjects of these concepts is sometimes very difficult to read, once the underlying mathematics is broken down into smaller steps, it all becomes clearer. The individual steps consist of very basic operations and should therefore be easy to understand. This section gives the reader a quick tutorial in the above concepts. A more experienced or mathematically versed reader may wish to skip this section. To the novice we recommend the detailed examples in APPENDIX I. The later sections contain brief examples to illustrate the steps of the algorithms used. We have found worked examples to be a great help in understanding the processes behind the algorithm and have included them for the reader's benefit.

2.1. Arbitrary Precision

The issue of controlling precision has been around a long as computing itself. Floating point arithmetic has the advantage of speed, but the drawback is that the result of a large number of calculations can be very inaccurate if not downright incorrect [30]. Another problem with floating point arithmetic is that there is only a finite amount of space to store a sometimes infinite number (say, an irrational number) and there will inevitably be truncation or rounding. Therefore, over a large number of calculations, we accumulate a round-off error, potentially resulting in a result that does not have the required precision. There are several different ways this can be remedied. In constraint programming systems, the desired precision is usually achieved using interval arithmetic. The result is not given by a single number, but by an

approximation: only the bounds of the interval in which the final answer lies is known. Although it is possible to control the precision of the outcome using interval arithmetic, it has the disadvantage that, should a result prove too inaccurate the whole set of calculations would have to be repeated from scratch until a sufficiently small interval is achieved.

A number represented as a continued fraction comes with a natural error bound. The continued fraction expansion of a number can be terminated at any partial quotient. The oscillatory nature of these successive approximations provides us with a single value for the error by taking the difference between the current and previous partial quotients at any given stage of refinement. In addition to that, continued fractions with recurring patterns can be represented exactly by rational trees.

One way to control precision is to perform a calculation to a certain accuracy and then truncate the result. This intermediate result is then used in further calculations and its accuracy determines the precision of the final result. This type of arithmetic is known as variable precision arithmetic. The responsibility for generating a sufficiently precise answer lies with the user, who decides how precise intermediate results will be by specifying the desired accuracy for individual assignments in the calculations. Macsyma, Maple, Mathematica and Par-gp are all examples of symbolic systems that work in this way [13].

Alternatively, the responsibility for the precision can lie with the system. The system determines the necessary accuracy of intermediate results and makes sure the final result has the desired precision. This is an extension to variable precision arithmetic known as exact arithmetic. The algorithm we will be using falls into the latter category: the user specifies only the desired precision for the final result. The system ensures it is achieved by controlling the precision of the input and no round-off error is accumulated over the calculations. It does not truncate to rational numbers early on in the process [1], so no information is lost during calculation.

2.2. Continued Fractions

Continued fractions have been around for centuries. they have enabled mathematicians such as Gauss or Laplace to guarantee a high level of precision in their calculations long before the advent of computers [22].

More recently, the areas continued fractions have been most commonly used in are in conjunction with floating point implementations of real numbers on the one hand and, on the other hand, in continued fraction representations of real numbers. Continued fractions still provided an elegant representation of numbers, both rational and irrational [11] and have therefore often been the basis of exact real arithmetic, especially in the world of functional programming. Rational numbers can be represented by finite continued fractions and irrational numbers can be represented by infinite continued fractions.

Gosper [12] illustrates this with the example of finding the number of centimetres per inch. The continued fraction of 2.54 is [2 1 1 5 1 3], or $2.54 = 2 + 1/(1 + 1/(1 + 1/(5 + 1/(1 + 1/3))))$. Gosper goes on to show that the number of inches per metre is [39 2 1 2 2 1 4] , "much nicer than 39.(370078740157480314960629921259842519685039) where the part in parentheses repeats forever."

Automating continued fraction arithmetic has several advantages.

In [1], Gosper summarised them as follows:

• unlimited significance arithmetic without multiprecision

multiplication or division

• built in error analysis

• easy computation of algebraic functions

• no unnecessary computations

• no discarding of information (eg roundoff, truncation)

• reversibility of computations

• terms of the answer start to come out right away

• each operation requests terms only when necessary

• no calculation is performed unnecessarily

Continued fractions are usually written in the form

$$x = x_0 + \cfrac{a_0}{x_1 + \cfrac{a_1}{x_2 + \cfrac{a_2}{\ddots \; x_{k-1} + \cfrac{a_{k-1}}{x_k}}}}$$

We use so-called regular continued fractions, where all the a_i coefficients are 1, and all the x_i coefficients (except perhaps x_0) are at least 1.

Regular continued fractions are usually written $x = [x_0, x_1, x_2, \ldots, x_k]$ or in the following form:

$$x = x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{\ddots\; x_{k-1} + \cfrac{1}{x_k}}}}$$

The partial quotients in the continued fraction above would therefore be

$$r = x_0$$

$$r_0 = x_1 + \frac{1}{r_1}$$

$$r_1 = r_0 + \frac{1}{r_2}$$

$$\vdots$$

$$r_{k-1} = x_{k-1} + \frac{1}{r_k}$$

$$r_k = x_k$$

The partial quotients $\frac{p_n}{q_n}$ of a continued fraction are successive refinements of the fraction.

We generate continued fraction expansions according to the algorithm Vardi presents in [36]. This is based on Euclid's algorithm for finding the greatest common divisor of two numbers [5]. The algorithm is summarised below:

$$f(q) = [Int]$$

$$q = \frac{1}{0} \Rightarrow [\quad]$$

$$o/w \quad [\lfloor q \rfloor] * f\left(\frac{1}{q - \lfloor q \rfloor}\right)$$

In [5] Davenport a geometrical interpretation of continued fractions. The partial quotients $\frac{P_i}{Q_i}$, or convergents, of a continued fraction x can be plotted on a graph. Each point X_i is given by the cartesian coordinates (P_i, Q_i). Each pair P_i and Q_i must be co-prime. If they are not, the GCD is determined and the rational simplified. Successive partial quotients seem snap to an integral grid, as they each represent a pair of integers whose quotient is an approximation of the original number, or principal convergent $\frac{P_k}{Q_k}$. The oscillatory nature of the partial quotients $\frac{1}{1}, \frac{2}{1}, \frac{5}{3}$ and $\frac{7}{4}$ of $\sqrt{3} = [1;1\,2]$ is clearly visible on the following diagram

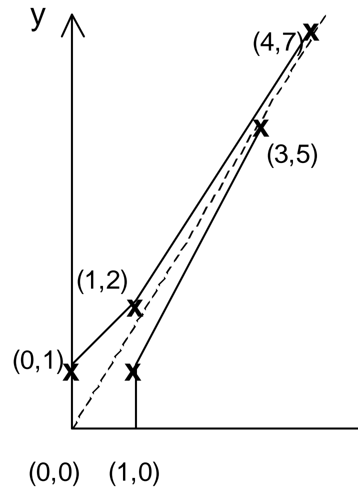

Figure 2-1. Geometrical representation of a continued fraction

2.3. Delay Declarations

The delay declaration is an important feature of logic programming systems that is rarely used in this way. In the Prolog implementation of continued fraction arithmetic, delay declarations are a key control mechanism. that enables us to compute numeric results lazily to the required degree of precision. In Sicstus Prolog, delay declarations are written in the form

`:- block predicate( ?, -)` for predicates with two arguments where the predicate waits for the second argument. For every argument of the predicate there must either be a "?" or a "-":

-- the argument that must be instantiated

? any other argument

In the following block of code, we have a simple example of a delay declaration in action. In this case, the predicate has three arguments and has to wait for the first.

```
:- block add_2_to_x(-,?).
add_2_to_x(X, Result) :-
     Result is X + 2.
```

The predicate `add_2_to_x` /2 will wait for its first argument to be instantiated. Only then will it begin i.e. add 2 to the value of X.

Sometimes it is necessary to wait for more than one argument before the evaluation can begin. In this case the predicate is preceded by two or more delay declarations. The following block of code calculates the sum of two numbers X and Y. In this scenario we have decided that, before it does so,

both X and Y should be instantiated. To ensure this is the case, we have used two block declarations, waiting for X and Y, respectively.

```
:- block add_X_Y(-,?,?).
:- block add_X_Y (?,-,?).
add_X_Y (X,Y,Result) :-
    Result is X + Y.
```

Since both delay declarations are used in conjunction, they can also be written in the following form:

```
:- block add_X_Y(-,?,?), block
add_X_Y (?,-,?).
```

We have chosen to use the former for clarity. The form where the individual block declarations are listed one underneath the other is easier to read.

2.4. Related Work

This project brings together different areas of research. Although the history of continued fractions in general goes back to Ancient Greek mathematics [10], our attention has been directed mostly at the results of more recent research into techniques for computing with exact real numbers using continued fractions. Continued fractions are relevant to many areas, such as number theory, probability theory and the analysis of algorithms. Kornerup and Matula [21], who base their number theoretic work on Hardy and Wright, handle convergence of continued fractions containing leading zeros to ±∞ in a bit-wise manner, whereas Vuillemin takes a slightly different approach [23]. Vardi bases his work on Vuillemin's system and presented pseudo code and Mathematica code for continued fraction expansions and continued fraction arithmetic in [37]. Unfortunately the algorithm for continued fraction expansion of $\frac{ax+b}{cx+d}$ contains an error. The conditions for the output of a continued fraction are supposedly that a, b, c and d must all be positive non-zero numbers and that $\lfloor \frac{a}{c} \rfloor$ must be the same as $\lfloor \frac{ax+b}{cx+d} \rfloor$. We have found that the second output condition should read "$\lfloor \frac{a}{c} \rfloor = \lfloor \frac{b}{d} \rfloor$". The more recent work [10] contains the same error.

An alternative to regular continued fractions is to take a continued fraction by "excess" [10]. The integral part is always positive, but the rest is subtracted, rather than added. The resulting partial quotients would therefore all be over-estimates of the number represented by the continued fraction.

In [1] and [12], both unpublished, Gosper presents continued fraction arithmetic algorithms that are independent of programming paradigms. He bases these on the theory behind the basic operations addition, subtraction, multiplication and division as originally described by Hall in 1947 [15]. Knuth gives a brief description of the algorithm in [20]. Continued fraction expansions as a framework for exact real computer arithmetic in functional programming were proposed by Gosper (Lisp), developed by Peyton-Jones [30], Lester (Miranda) [23] and Vuillemin (Miranda) and advanced more recently by Kornerup et al. Infinite composition of linear fractional transformations (also known as homographies or Möbius transformations) generalises the other two frameworks demonstrated by Vuilemin [39]. In [39], his difficult but fundamentally important work, Vuillemin describes redundant digital representations, tensors and Möbius transformations for continued fraction expansions, and presents an algorithm for the calculation of powers of numbers using continued fractions. His work has been extended by Heckmann, who has defined algorithms for the calculation of trigonometric functions and square roots. Over two centuries ago, Lambert and Euler developed algorithms for calculating tan and exp, respectively. One of Lester's contributions is the application of continued fractions in statistics [22] to evaluate standard statistical distribution functions.

Exact real arithmetic systems have been implemented in a variety of ways, including intervals, linear maps and linear fractional transformations. The latter have the advantage over linear maps that the continued fractions for many mathematical functions can be used almost directly [30]. It is for this reason that numerous researchers, among them Nielsen and Kornerup, investigate infinite compositions of Möbius transformations.

Gibbons' work on streaming representation-changers is closely related. Unfortunately, Gibbons has not published his work on binary rational functions $\frac{axy+bx+cy+d}{exy+fx+gy+h}$ of continued fractions that combine two arguments into one result [11].

In this project we have considered only linear fractional transformations, as we were concerned only with basic arithmetic operations. However, quadratic fractional transformations apparently have considerable merit for computing x^2 and transcendental functions although the composition

of a quadratic fractional transformation and a linear fractional transformation is not particularly elegant [31].

The quadratic fractional form is written

$$\frac{ax^2+bx+c}{dx^2+ex+f}$$

and can be expressed as a matrix $\begin{pmatrix} a\ b\ c \\ d\ e\ f \end{pmatrix}$.

Exact real arithmetic systems are, of course, not just limited to functional programming. We owe a JAVA implementation of constructive reals to Boehm [2]. ICReals has been developed at Imperial College for the C programming language. XR is a package for python or C++, with a functional extension FC++. Like iRRAM, also for C++, and many others, these packages give a look and feel of exact real arithmetic. However, much of the recent research into basing exact real arithmetic on homographies has been the focus of researchers in functional programming. This is partly due to the availability of lazy streams in functional programming. This concept can now be extended to logic programming, where current systems are all equipped to calculate numbers to an arbitrary level of precision simply by exploiting features such as streaming, rational trees and delay declarations, and therefore need not rely on interval arithmetic anymore.

3. Aims

The aim of this project was to find an alternative to interval arithmetic for arbitrary precision arithmetic in logic programming. There was a hunch that continued fractions may be involved. There had been similar work in the area of functional programming which needed to be synthesized and adapted to the logic programming paradigm. Then it was a matter of trying to implement parts of an algorithm in Prolog to see whether the approach could be used in logic programming.

4. Continued Fractions as Digit Streams

Digit streams are a useful representation of numbers because we can control the precision of a number by controlling the depth of the stream. Streams are a common feature of second-generation logic programming systems, so we do not have to implement them specially. Streams are very much like lists in Prolog. The difference is that streams are open ended.

Consider the stream `[X,Y,Z | Rest]`. Instantiating `Rest` to an open ended stream, for example, assigning `Rest=[A,B,C | Rest2]` and thereby introducing three more terms `A`, `B`, and `C`, will refine the representation to `[X, Y, Z, A, B, C | Rest2]` whilst preserving the ability for further refinement.

To generate a digit stream to represent a number, we find the continued fraction expansion of that number.

The complete code for the continued fraction expansion of a number can be found in APPENDIX IV. In short, a number is transformed into a digit stream by removing the integral part of the number, and sending it to an output stream, and then doing the same to the reciprocal of the non-integral rest.

Prolog is particularly well suited to this form of number representation. Not every number can be represented exactly by a finite stream. Take irrational numbers, for instance. They have to be represented by an infinite stream. Very often there is a recurring pattern. Continued fractions with such patterns are called periodic continued fractions. A purely periodic fraction is one where the period begins immediately and is not preceded by any other terms. Prolog can handle this easily because of rational trees. It just loops back to the beginning of the period every time it reaches the end of the pattern.

The following diagram shows the rational tree for the continued fraction expansion of $\sqrt{3}$. The digit stream representation is [1, 1, 2, 1, 2, 1, 2, …].

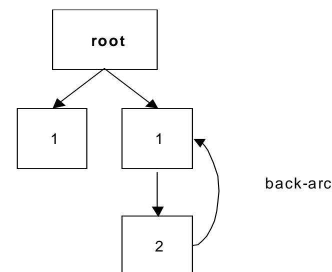

Figure 4-1. Rational tree for [1; 1, 2]

The desired precision is achieved by truncating the stream when the error no longer exceeds the ε determined by the user. The error is given by the difference between two successive partial quotients. For example, the difference between the fifth and sixth partial quotients of the Golden Ratio Φ is

$$\frac{13}{8}-\frac{8}{5}=\frac{1}{40}=0.025\,.$$

If ε is 0.025 or larger, the result is precise enough and the number can be built from the stream.

```
   expand_aux(R1,R2,Ss,Eps,R) :-
     subtract_rat(R1, R2, X),
     abs_rat(X, Y, Done),
     % set done flag!
expand_aux2(R2,Ss,Eps,R,Y,Done).

   %    wait    until    _Done    is
instantiated
   :- block expand_aux2(?,?,?,?,?,-
).
   % done flag!
   expand_aux2(R2,Ss,Eps,Rt,Y,_Done
) :-
        less_than_rat(Y, Eps) ->
                Result = R2;
                expand_stream(Ss,R2,
2,Eps,R).
```

5. Continued Fraction Arithmetic

To carry out binary operations, we use two input streams. The basic algorithm has been outlined in [12].

To work with the operations, it is easiest to see them as matrix transformations. These are quite simple and only involve basic operations on their coefficients.

Definition 5-1

$$blff_{A,E}(x,y) = \frac{axy+bx+cy+d}{exy+fx+gy+h},$$

where $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and $E = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$.

5.1. Two inputs

To put the matrix between the two streams, we have to represent the latter in matrix form too. As described in Section 2.2., each element of the stream can be represented by a matrix and the resulting matrices multiplied together, two at a time. Instead of just multiplying the matrices representing the stream's elements together, we alternate between the two streams and always pre-multiply the matrix coming from one stream and post-multiply the matrix coming from the other.

We have summarised the post-multiplication y in the following proposition. This equivalence can be found in the literature [1], the proof, however, is usually left as an exercise for the reader. We have included it in the APPENDIX III.

Proposition 5-1

$$blff_{A,E}(x, q+\tfrac{1}{y'}) = blff_{AQ,EQ}(x, y')$$

The pre-multiplication by x works similarly.

Proposition 5-2

$$blff_{A,E}(q+\tfrac{1}{x'}, y) = blff_{QA,QE}(x', y)$$

To begin, we need to know what numbers to put into the starting matrix. Then we have to let the matrices formed from the elements of the streams take turns.

This method corresponds roughly to the methods applied in most of the sources we consulted. To alternate between the streams, Vardi kept transposed the transformation matrix before every input [36]. Vuillemin merged the input streams, handling only one input stream from then on [39]. All these methods are variations of the same algorithm and look most different on paper. However, to Prolog the coefficients of the matrices are all just individual variables. Here they are called A,B,C,D,E,F,G, and H.

```
blff(A,B,C,D,E,F,G,H,S1,S2,R)
```

5.2. The operation

The following table lists the transformation matrices for the four basic arithmetic operations. We have given the details of how these are derived in APPENDIX II.

Table 1. Matrices representing the basic operations

Operation	Matrix
Addition	$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Subtraction	$\begin{pmatrix} 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Multiplication	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Division	$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

The algorithm continues to alternate between the two streams, emulating matrix multiplication, until either the required precision or the end of a stream is reached. When we reach the last element

of stream, it is represented by a vector $\begin{pmatrix} q \\ 1 \end{pmatrix}$, where q is the element of the stream. The result of multiplying a pair of 2x2-matrices with a vector is a 2x2 matrix. The elements of the remaining stream area then also expressed as vectors and the final result is a vector. The representations of the streams are still multiplied by the tensor as before: x is pre-multiplied and y is post-multiplied.

5.3. The final result

The answer is in the form of a stream of numbers again. To arrive at the decimal form, we build the number from this stream as we did in the single digit stream earlier. We merely reverse the process of generating a continued fraction expansion.

We do not have to wait for the answer until the end of the stream is reached. As stated earlier, we can truncate the input streams and translate the answer into decimal at any time. If more precision is required, we continue where we left off. The algorithm takes the final answer, in stream form, and continues until the new desired precision is reached. It is not necessary to perform all the calculations again from scratch. The algorithm can continue from the most recent digit stream and matrix coefficients and just iterate further from there.

Only the decimal representation of the number is returned at the end of the calculations, as it is easier for the user to understand. The result, however, is not generated right at the end of the algorithm. The digit stream is generated continuously throughout the process. The algorithm checks for the output condition after every input. If this condition is met, the value of the output digit is determined and new coefficients for the transformation matrices calculated.

5.4. Output

This condition posed difficulties at first, as the calculations never seemed to work after that point. In the end, we realised we had found an error in the literature and proceeded. The correct condition is that the floored ratios between the corresponding coefficients of the matrix pair have the same integer values:

$$\left\lfloor \frac{a}{e} \right\rfloor = \left\lfloor \frac{b}{f} \right\rfloor = \left\lfloor \frac{c}{g} \right\rfloor = \left\lfloor \frac{d}{h} \right\rfloor = q$$

The reasoning behind this is that we have a multiple of $exy + fx + gy + h$ in the top line, with bits left over. So we remove the integral multiple and repeat the entire process for the reciprocal of the remainder. The latter is smaller than 1, but the reciprocal would be larger. This is analogous to Euclid's algorithm for finding the GCD of two numbers. We have summarised this as follows:

Proposition 5-3

$$blff_{A,E}(x, y) - q = blff_{E,A-qE}(x, y)$$

The new tensor coefficients are, therefore,

$$\left(\begin{pmatrix} e & f \\ g & h \end{pmatrix}, \begin{pmatrix} a-qe & b-qf \\ c-qg & d-qh \end{pmatrix} \right).$$

The proof is given in APPENDIX III.

In Prolog this is easily done. We do not need any complicated data structures. We can evaluate and assign the new values of the matrix coefficients by way of `is` statements.

```
A1 is E,
B1 is F,
C1 is G,
D1 is H,
E1 is A-Q*E,
F1 is B-Q*F,
G1 is C-Q*G,
H1 is D-Q*H,
```

Sending a value to the output stream in Prolog is equally easily done. We add the digit `Q` to the head of the stream `Result`, just as we would add it to the head of a list.

```
Out = [Q | Result],
```

The understanding of the individual steps does not require great mathematical expertise. One of the achievements of this project was the breaking down of this algorithm into clear and simple steps and adapting it to logic programming.

6. Problems Encountered

One of the hurdles encountered early on in this project was the important yet extremely difficult paper by Vuillemin [39]. We tried out examples and looked for other sources. A lot of the literature was written for an audience with a high level of expertise. It took time to reach a level of mathematical ability sufficient to digest the literature. We tried to contact some of the authors but received no response before the end of the project.

Some sources, though reviewed and published, contained errors. One error in particular made it almost impossible to proceed. Eventually, when we had gained some confidence in the area, we uncovered and corrected it.

We found that some operations have to be implemented differently. We noticed that when we tried subtraction, the output was completely nonsensical. Knowing what we do about how numbers are expressed as continued fractions, we attribute this to the representation of zero and the fact that we did not allow negative numbers in the continued fraction expansions.

The theoretical background to this algorithm has been developed and published in the past. It was, however, necessary to distil the parts that are relevant to this project and really understand them. Then it was a matter of filling in gaps and expressing the theory in a way "that would be comprehensible to an average computer science graduate". This was one of the project requirements. Often, we had to retrace the steps of our predecessors in the field blindly, as some of the linking information was missing. We attribute this to the constraints on the length of a publication and the fact that the experts may have deemed some of the information we needed too obvious.

7. Conclusions

The project set out with an apparently simple goal but we soon found that there was more research to be done than initially expected.

The main objective was to find an alternative to interval arithmetic for arbitrary arithmetic in logic programming. This objective has been achieved. The implementation has served as a sounding board for the theory. The algorithm described here calculates the sum or product of a pair of streams accurately. It seems there is a flaw in the method for division which, given some more time, could be found.

The issue with subtraction is clear: negative numbers can occur, so an alternative to $\mathbb{N}$-fractions must be used.

Similar work has already been done in the field of functional programming. However, in logic programming algorithms for continued fraction arithmetic to calculate the sum or product of a pair of numbers, rational or irrational, to arbitrary precision do not seem to have been implemented yet. One of the achievements of this project is the unusual use of delay declarations.

The literature was very scattered and one of the tasks was to consolidate it. It is sometimes difficult to know where to begin and, later, where to stop. We could not follow up every reference, as some consisted of unpublished material, email correspondence or internal notes. Published material sometimes contained errors. There was one unexpected and fundamental error that caused a significant delay, but we are fortunate to have been able to spot it at all. All the time invested in understanding the mathematics resulted in finding a suitable algorithm to implement in logic programming. It is now clear that it is possible to implement arbitrary precision arithmetic in logic programming using continued fractions to represent numbers and delay declarations for control.

There is much of scope for further work. The algorithm could be extended to other mathematical operations, such as division and subtraction. Alone for subtraction, there is much work to be done, as the algorithm has to be extended to handle non-positive digits. One solution might be to use $\mathbb{E}$-fractions instead of $\mathbb{N}$-fractions. Related issues include multiple representations of numbers, especially zero. Delay declarations could result in deadlock. Further work could examine methods to prevent deadlock. Finally, it would be interesting to see whether transcendental functions can be implemented in a similar manner. The challenge is there, it is only a matter of taking it up.

8. Acknowledgements

Our thanks go to Dr. Andy King, the project supervisor, for his patience and support throughout the project. We are grateful, too, to Prof. Simon Thompson, Dr. Keith Hanna and Dr. Stefan Kahrs for their invaluable feedback.

9. Bibliography

[1] M. Beeler, R.W. Gosper and R. Schroeppel, "HAKMEM", *A.I. Lab Memo # 239*, M.I.T. 1972.

[2] H. Boehm and R. Cartwright, "Exact Real Arithmetic: Formulating real numbers as functions", *Research Topics in Functional Programming*, Addison-Wesley 1990, pp. 43–64.

[3] R.P. Brent, A.J. van der Poorten and H.J.J. teRiele, "A Comparative Study of Algorithms for Computing Continued Fractions for Algebraic Numbers", *International Symposium on Algorithmic Number Theory*, Springer-Verlag 1997, pp. 35–48.

[4] Chrystal, G., *Algebra - an Elementary Text-book for the Higher Classes of Secondary schools and for Colleges, Part II*, Chelsea Publishing Company 1964.

[5] Davenport, H., *The Higher Arithmetic*, Cambridge University Press 2003.

[6] A. Edalat, "Domains for Computation in Mathematics, Physics and Exact Real Arithmetic", *Bulletin of Symbolic Logic*, **3**(4) (1997), pp. 37–44.

[7] A. Edalat and R. Heckmann, "Computing with Real Numbers", *International Summer School On Applied Semantics*, LNCS 2395, Springer-Verlag (2000), pp. 193–267.

[8] Bratko, I., *PROLOG - Programming for Artificial Intelligence*, Addison-Wesley 1990.

[9] Clocksin, W.F., and C.S. Mellish, *Programming in Prolog*, Springer-Verlag 1994.

[10] P. Flajolet, B. Vallée and I. Vardi, "Continued Fractions from Euclid to the present day", www.lix.polytechnique.fr/Labo/Ilan.Vardi/publications.html.

[11] J. Gibbons, "Streaming representation-changers", *International Conference on Mathematics of Program Construction,* LNCS 3125, Springer-Verlag (2004), pp. 1–27.

[12] R.W. Gosper, "Continued Fraction Arithmetic", preprint 1976.

[13] P. Gowland and D. Lester, "A Survey of Exact Arithmetic Implementations", *Computability and Complexity in Analysis*, LNCS 2064, Springer-Verlag (2001), pp. 30–47.

[14] Gregory, S., *Parallel Logic Programming in PARLOG*, Addison-Wesley (1987), pp. 103–107.

[15] M. Hall, Jr., "On the sum and product of continued fractions", *Annals of Mathematics,* **48**(4) (1947), pp. 966–993.

[16] R. Heckmann, "Contractivity of Linear Fractional Transformations", *Third Real Numbers and Computers Conference, Theoretical Computer Science*, **279**(1-2) (2002), pp. 65–82.

[17] ILOG Solver, 5.3 ed. Paris: ILOG Inc., http://www.ilog.com/, 2003.

[18] Khinchin, A.Y., *Continued Fractions*, University of Chicago Press, Chicago 1964.

[19] R. Knott, "An Introduction to the Continued Fraction", http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/cfINTRO.html.

[20] Knuth, D.E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Addison–Wesley 1981.

[21] P. Kornerup and D.W. Matula, "Finite-Precision Lexicographic Continue Fraction Number Systems", *Symposium on Computer Arithmetic*, IEEE (1995), pp. 207–214.

[22] D. Lester, "Exact Statistics and Continued Fractions", *Journal of Universal Computer Science*, **1**(7) (1995), pp. 504–513.

[23] D. Lester, "Effective Continued Fractions", IEEE *Symposium on Computer Arithmetic*, IEEE (2002), pp. 65–82.

[24] Moore, R.E., *Interval Analysis*, Prentice-Hall 1966.

[25] N.Th. Mueller, "The iRRAM: Exact Arithmetic in C++", *Selected Papers from the 4th International Workshop on Computability and Complexity in Analysis*, LNCS 2064**,** Springer-Verlag (2001), pp. 222-252.

[26] A. Nielsen and P. Kornerup, "MSB-First Digit Serial Arithmetic"*, Journal of Universal Computer Science*, **1**(7) (1995), p. 527–547.

[27] M. Niqui and Y. Bertot, "QArith: Coq Formalisation of Lazy Rational Arithmetic", *Types for Proofs and Programs: Third International Workshop*, LNCS 3085, Springer-Verlag (2004), pp. 309 – 323.

[28] W.J. Older, F. Benhamou, "Programming in CLP(BNR) ", *Principles and Practice of Constraint Programming*, Springer-Verlag (1993), pp. 228-238.

[29] Olds, C.D., *Continued Fractions*, Random House 1963.

[30] P.J. Potts, A. Edalat, and M.H. Escardó, "Semantics of Exact Real Arithmetic", *IEEE Symposium on Logic in ComputerScience*, IEEE 1997, pp. 248–257.

[31] Potts, P.J., "Exact Real Arithmetic using Möbius Transformations", PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, 1998.

[32] C. Rössner and C.P. Schnorr, "An Optimal, Stable Continued Fraction Algorithms for Arbitrary Dimension", *Integer Programming and Combinatorial Optimization 1996*, Lecture Notes in Computer Science 1084, Springer-Verlag (1996), pp. 31–43.

[33] M. Teichmann, "Complexity of Some Elementary Operations on Continued Fractions", http://citeseer.ist.psu.edu/28745.html.

[34] P. Van Hentenryck, D. McAllester, D. Kapur, "Solving Polynomial Systems Using a Branch and Prune Approach", *SIAM Journal on Numerical Analysis*, **34**(2) (1997), pp. 797–827.

[35] P. Van Hentenryck, "A gentle introduction to Numerica", *Artificial Intelligence*, 103(1-2) (1998), pp. 209-235.

[36] I. Vardi, "Code and Pseudo Code", *The Mathematica Journal*, **6**(2) (1996), pp. 66–71.

[37] Vorob'ev, N.N., *Fibonacci Numbers*, Birkhauser 2003.

[38] J. Vuillemin, "Arithmétique Réelle Exacte par les Fractions Continues", *Rapports de Recherche # 760*, INRIA 1987.

[39] J. Vuillemin, "Exact Real Computer Arithmetic with Continued Fractions", *IEEE Transactions on Computers*, **39**(8) (1990), pp. 1087–1105.

[40] Wall, H.S., *Analytic Theory of Continued Fractions*, Van Nostrand, Inc 1948.

10. APPENDIX I - Worked Examples

10.1. Euclid's Algorithm

Euclid's algorithm is used to calculate the greatest common divisor (GDC), also known as the highest common factor, of two numbers. In the following example, we show how the recursive subtraction algorithm finds the GCD of 120 and 504.

The Prolog program looks like this:

```
gcd(X, X, X).
gcd(X, Y, Result) :-
   X<Y,
   Y1 is Y-X,
   gcd(X, Y1, Result).
gcd(X, Y, Result) :-
   Y>X,
   gcd(Y, X, Result).
```

Typing "`|?- gcd(120, 504, D).`" into Sicstus would result in the following calculations:

```
gcd(120,504,D)
 120<504,
Y1 is 504-120=384,
Call gcd(120,384,D),
120<384,
Y2 is 384-120=264,
Call gcd(120,264,D),
120<264.
Y3 is 264-120 = 144,
Call gcd(120,144,D),
120<144,
Y4 is 144-120=24,
Call(gcd120,24,D),
24<120,
Call gcd(24,120,D),
24<120,
Y5 is 120-24=96,
Call gcd(24,96,D),
24<96,
Y6 is 96-24=72,
Call gcd(24,72,D),
24<72,
Y7 is 72-24=48,
Call gcd(24,48,D),
24<48,
Y1 is 48-24=24,
Call gcd(24,24,D),
D=24.
```

This final value is propagated right up to the top, giving the final result "`D=24`". As we know,

$120 = 2*2*2*3*5 = 24*5$

$504 = 2*2*2*3*3*7 = 24*21$

The above steps are not Prolog output, but merely an illustration of the steps involved in the calculation.

We used this algorithm, but included predicates to take the absolute values of X and Y, so the comparisons would only be between the magnitudes of the numbers.

```
gcd(X, Y, Result) :-
      X < 0,
      X1 is 0-X,
      gcd(X1, Y, Result).
gcd(X, Y, Result) :-
      Y < 0,
      Y1 is 0-Y,
      gcd(X, Y1, Result).
```

10.2. Continued Fraction Expansion

Regular continued fractions are usually written $x = [x_0, x_1, x_2, \ldots, x_k]$ or in the following form:

$$x = x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{\ddots\, x_{k-1} + \cfrac{1}{x_k}}}}$$

The partial quotients in the continued fraction above would therefore be

$$\mathrm{r} = \mathrm{x}_0$$

$$\mathrm{r}_0 = \mathrm{x}_1 + \frac{1}{\mathrm{r}_1}$$

$$\mathrm{r}_1 = \mathrm{r}_0 + \frac{1}{\mathrm{r}_2}$$

$$\vdots$$

$$\mathrm{r}_{k\text{-}1} = \mathrm{x}_{k\text{-}1} + \frac{1}{\mathrm{r}_k}$$

$$\mathrm{r}_k = \mathrm{x}_k$$

The partial quotients $\frac{p_n}{q_n}$ of a continued fraction are successive refinements of the fraction. Here is a simple example.

$$x = \frac{p}{q} = \frac{41}{11}$$

The first term x_0 in the continued fraction x is found using Euclid's algorithm for finding the GCD of two numbers.

$$\frac{p}{q} = \frac{41}{11} = 3 + \frac{8}{11} = 3 + \frac{1}{\frac{11}{8}}$$

We now know $x_0 = 3$, so our first partial quotient $\frac{p_0}{q_0}$ is also 3. The partial quotients of a continued fraction converge toward the exact value, so this is the least precise value of x. Also, $\frac{p_0}{q_0} < x$.

The next term is calculated in the same manner.

$$r_1 = \frac{11}{8} = 1 + \frac{3}{8} = 1 + \frac{1}{\frac{8}{3}}$$

$x_1 = 1$. From the terms we have found so far, we can calculate a closer approximation of the value of x.

$$\frac{p_1}{q_1} = 3 + \frac{1}{1} = 3 + 1 = 4 > x$$

We continued calculating the terms of the continued fraction until we reach a remainder 0.

$$r_2 = \frac{8}{3} = 2 + \frac{2}{3} = 2 + \frac{1}{\frac{3}{2}}$$

So $x_2 = 2$ and

$$\frac{p_2}{q_2} = 3 + \cfrac{1}{1 + \cfrac{1}{2}} = \frac{11}{3} < x$$

For the third term,

$$r_3 = \frac{3}{2} = 1 + \frac{1}{2} = 1 + \frac{1}{\frac{2}{1}}$$

So $x_3 = 1$ and

$\frac{p_3}{q_3}$
$$= 3 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1}}}$$
$$= \frac{15}{4} > x$$

Last, but not least,

r_4
$$= \tfrac{2}{1}$$
$$= 2.$$

So our last term is $x_4 = 2$ and

$\frac{p_4}{q_4}$
$$= 3 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2+0}}}}$$
$$= \frac{41}{11} = x$$

The continued fraction has converged at $n = 4$.

The even partial fractions always lie below the exact value and the odd partial fractions always lie above it. The continued fraction can also be represented as a digit stream consisting of its terms

$$[\,3, 1, 2, 1, 2\,].$$

In this example the partial quotients $\frac{p_n}{q_n}$, or convergents, obtained by evaluating the continued fraction to successively deeper terms are $3, 4, \frac{11}{3}, \frac{15}{4}$ and $\frac{41}{11}$.

10.3. Continued Fraction Arithmetic

Written documents, detailed diagrams and tables are often better presented on paper. The CD-ROM then contains items which will only be read briefly or items which contain large amounts of data.

11. APPENDIX II - From Fractional Form to Matrices

The bilinear fractional form is written in the following way:

$$\frac{axy + bx + cy + d}{exy + fx + gy + h}$$

The addition of two numbers x and y can be represented by the bilinear fractional form:

$$x + y = \frac{0xy + 1x + 1y + 0}{0xy + 0x + 0y + 1}$$

Taking only the coefficients would give the shorthand notation:

$$\frac{0\,1\,1\,0}{0\,0\,0\,1}$$

In the table in SECTION we used a simple notation, where the top row of the 2x4-matrix represents the top line of the fraction and the bottom row of the matrix represents the bottom:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

As a tensor, or pair of matrices, the top line is represented by the first matrix and the bottom line is represented by the second, giving:

$$\left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right)$$

This above notation is commonly used in the literature. We use the same for our calculations.

We can repeat the process for the other basic operations. For subtraction we have

$$x - y = \frac{0xy + 1x - 1y + 0}{0xy + 0x + 0y + 1}$$

$$\frac{0\,1\,1\,0}{0\,0\,0\,1}$$

$$\begin{pmatrix} 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\left(\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right)$$

Multiplication is represented by:

$$xy = \frac{1xy + 0x + 0y + 0}{0xy + 0x + 0y + 1}$$

$$\frac{1\,0\,0\,0}{0\,0\,0\,1}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right)$$

Division can be expressed as:

$$\frac{x}{y} = \frac{0xy + 1x + 0y + 0}{0xy + 0x + 1y + 0}$$

$$\frac{0100}{0010}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\left(\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \right)$$

There is a difference between unary and binary operations, and transcendental functions are different again. Unary operations, such as multiplying a digit stream by a scalar, can be expressed as a linear fractional transformation (1-LFT) and require only a 2x2-matrix. Binary operations can be expressed as a bilinear fractional transformation (2-LFT) and involve 2x4-matrices. Some transcendental functions can be represented using a 2x3-matrix, which is another way of writing a quadratic fractional function (QFT). We have given a summary below.

Table 2. Transformation matrices

Name		size	Example
1-LFT	$\frac{ax+b}{cx+d}$	$2x2$	$\begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}$
2-LFT	$\frac{axy+bx+cy+d}{exy+fx+gy+h}$	$2x4$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
QFT	$\frac{ax^2+bx+c}{dx^2+ex+f}$	$2x3$	$\begin{pmatrix} 220 \\ 221 \end{pmatrix} \tan 1$

In the table above, the first example is similar to the 2x2 identity matrix and represents a scalar multiplication by 3. The second example represents the multiplication of two matrices. The last example represents sin 2.

12. APPENDIX III - Proofs

Here are the proofs for the propositions in the technical section about Continued Fraction Arithmetic.

$$\frac{A}{E} = \frac{axy + bx + cy + d}{exy + fx + gy + h}$$

The eight coefficients of this bilinear fractional form (blff) correspond to the eight coefficients of the matrix.

Definition 5-1

$$blff_{A,E}(x, y) = \frac{axy + bx + cy + d}{exy + fx + gy + h},$$

where $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and $E = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$.

12.1. Linear fractional form

The linear fractional form:

Proposition:

$\mathrm{lff}_A(q + \frac{1}{x'}) = \mathrm{lff}_{AQ}(x')$ where $Q = \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$

Proof:

$$\begin{aligned} & \mathrm{lff}_A(q + \frac{1}{x'}) \\ = \; & \frac{ax+b}{cx+d} \\ = \; & \frac{a(q+\frac{1}{x'})+b}{c(q+\frac{1}{x'})+d} \\ = \; & \frac{a(qx'+1)+bx'}{c(qx'+1)+dx'} \\ = \; & \frac{aqx'+a+bx'}{cqx'+c+dx'} \\ = \; & \frac{(aq+b)x'+a}{(cq+d)x'+c} \\ = \; & \mathrm{lff}_{\begin{pmatrix} aq+b & a \\ cq+d & c \end{pmatrix}}(x') \\ = \; & \mathrm{lff}_{AQ}(x') \end{aligned}$$

■

12.2. Post-multiplication

The following proposition and proof give the result of post-multiplication by y.

Proposition 5-1

$$blff_{A,E}(x, q + \tfrac{1}{y'}) = blff_{AQ,EQ}(x, y')$$

Proof:

$$\begin{aligned}
&\mathrm{blff}_{A,E}(x, q + \frac{1}{y'}) \\
&= \frac{axy + bx + cy + d}{exy + fx + gy + h} \\
&= \frac{ax(q + \frac{1}{y'}) + bx + c(q + \frac{1}{y'}) + d}{ex(q + \frac{1}{y'}) + fx + g(q + \frac{1}{y'}) + h} \\
&= \frac{ax(qy' + 1) + bxy' + c(qy' + 1) + dy'}{ex(qy' + 1) + fxy' + g(qy' + 1) + hy'} \\
&= \frac{aqxy' + ax + bxy' + cqy' + c + dy'}{eqxy' + ex + fxy' + gqy' + g + hy'} \\
&= \frac{(aq + b)xy' + ax + (cq + d)y' + c}{(eq + f)xy' + ex + (gq + h)y' + g} \\
&= \mathrm{blff}_{\begin{pmatrix} aq+b & a \\ cq+d & c \end{pmatrix}, \begin{pmatrix} eq+f & e \\ gq+h & g \end{pmatrix}}(x, y') \\
&= \mathrm{blff}_{AQ,EQ}(x, y')
\end{aligned}$$

■

12.3. Pre-multiplication

The following proposition and proof give the result of pre-multiplication by x.

Proposition 5-2

$$blff_{A,E}(q + \tfrac{1}{x'}, y) = blff_{QA,QE}(x', y)$$

Proof:

$$\begin{aligned}
&\mathrm{blff}_{A,E}(q + \frac{1}{x'}, y) \\
&= \frac{axy + bx + cy + d}{exy + fx + gy + h} \\
&= \frac{a(q + \frac{1}{x'})y + b(q + \frac{1}{x'}) + cy + d}{e(q + \frac{1}{x'})y + f(q + \frac{1}{x'}) + gy + h} \\
&= \frac{a(qx' + 1)y + b(qx' + 1)y' + cx'y + dx'}{e(qx' + 1)y + f(qx' + 1)y' + gx'y + hx'} \\
&= \frac{aqx'y + ay + bx'y + b + cx'y + dx'}{eqx'y + ey + fx'y + f + gx'y + hx'} \\
&= \frac{(aq + c)x'y + (bq + d)x' + ay + b}{(eq + g)xy' + (fq + h)x' + ey + f} \\
&= \mathrm{blff}_{\begin{pmatrix} aq+c & bq+d \\ a & b \end{pmatrix}, \begin{pmatrix} eq+g & fq+h \\ e & f \end{pmatrix}}(x, y') \\
&= \mathrm{blff}_{QA,QE}(x', y)
\end{aligned}$$

■

12.4. New transformation after output

The following proposition and proof give the new tensor coefficients after output.

Proposition 5-1

$$blff_{A,E}(x, y) - q = blff_{E,A-qE}(x, y)$$

Proof:

$$\begin{aligned}
&\mathrm{blff}_{A,E}(x, y) - \lfloor \frac{a}{e} \rfloor \\
&= \frac{axy + bx + cy + d}{exy + fx + gy + h} - \lfloor \frac{a}{e} \rfloor \\
&= \frac{axy + bx + cy + d - \lfloor \frac{a}{e} \rfloor (exy + fx + gy + h)}{exy + fx + gy + h} \\
&= \frac{(a - \lfloor \frac{a}{e} \rfloor e)xy + (b - \lfloor \frac{a}{e} \rfloor f)x + (c - \lfloor \frac{a}{e} \rfloor g)y + d - \lfloor \frac{a}{e} \rfloor h}{exy + fx + gy + h} \\
&= \frac{(a - q * e)xy + (b - q * f)x + (c - q * g)y + d - q * h}{exy + fx + gy + h} \\
&= \mathrm{blff}_{A-qE,E}(x, y) \text{ where } q = \lfloor \frac{a}{e} \rfloor = \lfloor \frac{b}{f} \rfloor = \lfloor \frac{b}{f} \rfloor = \lfloor \frac{b}{f} \rfloor
\end{aligned}$$

$$\begin{aligned}
&\mathrm{blff}_{E,A-qE}(x, y) \\
&= \frac{exy + fx + gy + h}{axy + bx + cy + d - q(exy + fx + gy + h)} \\
&= \frac{1}{\frac{axy + bx + cy + d - q(exy + fx + gy + h)}{exy + fx + gy + h}} \\
&= \frac{1}{\mathrm{blff}_{E,A-qE}(x, y)} \\
&= \frac{1}{\mathrm{blff}_{A-qE,E}(y, y)}
\end{aligned}$$

■

12.5. Last element in a stream

The following proposition and proof describe the transition from the bilinear fractional form to linear fractional form when the last element of a stream is reached.

Proposition:

$$\mathrm{blff}_{A,E}(x, q) = \mathrm{lff}_{A'}(x) \text{ where } A' = \begin{pmatrix} q & 1 & 0 & 0 \\ 0 & 0 & q & 1 \end{pmatrix} \begin{pmatrix} A^T \\ E^T \end{pmatrix}$$

Proof:

$$\begin{aligned}
&\mathrm{blff}_{A,E}(x, q) \\
&= \frac{axy + bx + cy + d}{exy + fx + gy + h} \\
&= \frac{axq + bx + cq + d}{exq + fx + gq + h} \\
&= \frac{(aq + b)x + (cq + d)}{(eq + f)x + (gq + h)} \\
&= \mathrm{lff}_{\begin{pmatrix} aq+b & cq+d \\ eq+f & gq+h \end{pmatrix}}(x) \\
&= \mathrm{lff}_{A'}(x) \text{ where } A' = \begin{pmatrix} q & 1 & 0 & 0 \\ 0 & 0 & q & 1 \end{pmatrix} \begin{pmatrix} A^T \\ E^T \end{pmatrix}
\end{aligned}$$

■

13. APPENDIX IV - Prolog Code

We have listed the commented code in a one-column format to make it more readable.

13.1. Stream

```
% build the integer stream [ , 1] to depth N (Golden Ratio)
:- block integer_stream(-, ?).
integer_stream([], _N).
integer_stream([H | T], N) :-
    H = N,
    integer_stream(T, N).
```

13.2. Continued Fraction Expansion

```
% CF expansion using auxiliary
% need to compare successive partial quotients
% so pq_stream twice
expand_stream(Ss, Eps, Result) :-
    pq_stream(Ss, 1, R1),
    pq_stream(Ss, 2, R2),
    expand_aux(R1, R2, Ss, Eps, Result).

:- block expand_aux(-, ?, ?, ?, ?).
:- block expand_aux(?, -, ?, ?, ?).
expand_aux(R1, R2, Ss, Eps, Result) :-
    subtract_rat(R1, R2, X
    abs_rat(X, Y, Done),
     % set done flag!
    expand_aux2(R2, Ss, Eps, Result, Y, Done).

% wait until _Done is instantiated
:- block expand_aux2(?, ?, ?, ?, ?, -).
% done flag!
expand_aux2(R2, Ss, Eps, Result, Y, _Done) :-
    less_than_rat(Y, Eps) ->
           Result = R2;
           expand_stream(Ss, R2, 2, Eps, Result).

expand_stream(Ss, R1, N, Eps, Result) :-
    N1 is N + 1,
    pq_stream(Ss, N1, R2),
    expand_aux(R1, R2, Ss, N1, Eps, Result).

:- block expand_aux(?, -, ?, ?, ?, ?).
expand_aux(R1, R2, Ss, N1, Eps, Result) :-
    subtract_rat(R1, R2, X),
    abs_rat(X, Y, Done),
     % set done flag!
     expand_aux2(Ss, R2, N1, Eps, Result, Y, Done)

% wait until _Done is instantiated
:- block expand_aux2(?, ?, ?, ?, ?, ?, -).
% done flag!
```

```
expand_aux2(Ss, R2, N1, Eps, Result, Y, _Done) :-
    less_than_rat(Y, Eps) ->
            Result = R2;
            expand_stream(Ss, R2, N1, Eps, Result).

% test: check whether A < E
less_than_rat(A, E) :-
    subtract_rat(A, E, Result),
    negative_rat(Result).

% split the rational N-R into components N and R
:- block negative_rat(-).
negative_rat(N-R) :-
    negative_aux(N, R).

% test: check whether N/R is negative
% wait for numerator N
:- block negative_aux(-, ?).
% wait for denominator D
:- block negative_aux(?, -).
negative_aux(N, R) :-
    N*R < 0.

% split the rational N-D into components N and D
:- block abs_rat(-, ?, ?).
abs_rat(N-R, Result, Done) :-
    abs_aux(N, R, Result, Done).

% absolute value of a fraction N/D
% wait for numerator N
:- block abs_aux(-, ?, ?, ?).
% wait for denominator D
:- block abs_aux(?, -, ?, ?).
abs_aux(N, R, N1-R, Done) :-
    N*R < 0 ->
          N1 is 0-N,
          Done = true;
          N1 = N,
          Done = true.

:- block subtract_rat(-, ?, ?).
:- block subtract_rat(?, -, ?).
subtract_rat(N1-D1, N2-D2, Result) :-
    subtract_aux(N1, N2, D1, D2, Result).

:- block subtract_aux(-, ?, ?, ?, ?).
:- block subtract_aux(?, -, ?, ?, ?).
:- block subtract_aux(?, ?, -, ?, ?).
:- block subtract_aux(?, ?, ?, -, ?).
subtract_aux(N1, N2, D1, D2, Rat_Result):-
    NR is N1*D2 - N2*D1,
    DR is D1*D2,
    simplify_aux(NR, DR, Rat_Result).

% build partial quotient from stream to depth N in rational form
pq_stream(Ss, N, Result) :-
    N == 1 ->
```

```
            Ss = [M | _],
            Result = M-1;
            N1 is N - 1,
            Ss = [S | Rest],
            pq_stream(Rest, N1, Partial_Quotient),
            my_is(S, Partial_Quotient, Result).

% split up rational(PQN-PQD) into PQN and PQD
:- block my_is(-, ?, ?).
:- block my_is(?, -, ?).
my_is(S, PQN-PQD, Result) :-
        is_aux(S, PQN, PQD, Result).

% build x = q + 1/x1 = (qN + D)/N for x=N/D
:- block is_aux(?, -, ?, ?).
:- block is_aux(?, ?, -, ?).
is_aux(S, PQN, PQD, Result):-
    N is S*PQN + PQD,
     % top subterms there => bottom ones (conv)
    Result = N-PQN.
```

13.3. Continued Fraction Arithmetic

```
% returns a stream to a depth 8.
run_this([S, T, U, V, W, X, Y, Z]):-
     % input stream S1 for x
    integer_stream(S1),
     % input stream S2 for y
    integer_stream(S2),
     % print heading to screen
    format("~nstep           in out  A B C D/ E F G H", []),
    format("~n-------------- -- ---  ----------------", []),
    plus(S1, S2,[S, T, U, V, W, X, Y, Z]).

% forward to blff, instantiating 8 variables

% (0xy + x + y + 0)/(0xy + 0x + 0y + 1)
plus(S1, S2, Result) :-
    blff(0, 1, 1, 0, 0, 0, 0, 1, S1, S2, Result).

% (0xy + x + (-1)y + 0)/(0xy + 0x + 0y + 1)
minus(S1, S2, Result) :-
    blff(0, 1, -1, 0, 0, 0, 0, 1, S1, S2, Result).

% (xy + 0x + 0y + 0)/(0xy + 0x + 0y + 1)
multiply(S1, S2, Result) :-
    blff(1, 0, 0, 0, 0, 0, 0, 1, S1, S2, Result).

% (0xy + x + 0y + 0)/(0xy + 0x + y + 0)
divide(S1, S2, Result) :-
    blff(0, 1, 0, 0, 0, 0, 1, 0, S1, S2, Result).

% ALGORITHM: pre-/post-multiply, check for integral part
% bilinear fractional form
% wait until Result is instantiated
:- block blff(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, -).
```

```
   % result = [], so do nothing
   blff(_, _, _, _, _, _, _, _, _, _, []).
   blff(A, B, C, D, E, F, G, H, S1, S2, Result) :-
       % stamp on S1
       S1 = [M | _],
       % print variables to screen
       format("~npre-multiply   ~w      ~w ~w ~w ~w/ ~w ~w ~w ~w", [M, A,
B, C, D, E, F, G, H]),
       % pre-multiply for y
       pre(A,B,C,D,E,F,G,H, M, A1,B1,C1,D1,E1,F1,G1,H1),
       % stamp on S2
       S2 = [N | _]
       % print variables to screen
       format("~npost-multiply  ~w      ~w ~w ~w ~w/ ~w ~w ~w ~w", [N, A1,
B1, C1, D1, E1, F1, G1, H1]),
       % post-multiply for x
       post(A1,B1,C1,D1,E1,F1,G1,H1,N,A2,B2,C2,D2,E2,F2,G2,H2),
       test_for_output(A2, B2, C2, D2, E2, F2, G2, H2, S1, S2, Result).

   % re test: if A/E = floor(A/E), then
   %  Euclidean step will result in a zero, so should FAIL test
   % test: if corresponding arguments of A and E give the same ratios,
   %  output the integral part of that value
   test_for_output(A, B, C, D, E, F, G, H, S1, S2, Out):-
       (A \== 0,
       B \== 0,
       C \== 0,
       D \== 0,
       E \== 0,
       F \== 0,
       G \== 0,
       H \== 0,
       % if all the floors the same
       integer(floor(A/E)) =:= integer(floor(B/F)),
       integer(floor(B/F)) =:= integer(floor(C/G)),
       integer(floor(C/G)) =:= integer(floor(D/H))) ->
              % get a single common integer value
              Q is integer(floor(A/E)),
              % print to screen
              format("~nEuclidean step   ~w     ~w ~w ~w ~w/ ~w ~w ~w ~w",
[Q, A, B, C, D, E, F, G, H]),
              % output to head of list
              Out = [Q | Result],
              A1 is E,
              B1 is F,
              C1 is G,
              D1 is H,
              E1 is A-Q*E,
              F1 is B-Q*F,
              G1 is C-Q*G,
              H1 is D-Q*H,
             % call blff with new 8 values (A-QE)
              blff(A1,B1,C1,D1,E1,F1,G1,H1, S1,S2, Result);
             % recursive call to blff here
              blff(A, B, C, D, E, F, G, H, S1, S2, Out).

   % wait until all 8 are instantiated
```

```
:- block post(-,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,-,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,-,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,?,-,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,?,?,-,?,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,?,?,?,-,?,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,?,?,?,?,-,?,?,?,?,?,?,?,?,?,?).
:- block post(?,?,?,?,?,?,?,-,?,?,?,?,?,?,?,?,?).
% wait until M is instantiated
:- block post(?,?,?,?,?,?,?,?,-,?,?,?,?,?,?,?,?).
post(A, B, C, D, E, F, G, H, M, A1, B1, C1, D1, E1, F1, G1, H1) :-
    A1 is A*M+B,
    B1 is A,
    C1 is C*M+D,
    D1 is C,
    E1 is E*M+F,
    F1 is E,
    G1 is G*M+H,
    H1 is G.

% wait until all 8 are instantiated
:- block pre(-,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,-,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,-,?,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,?,-,?,?,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,?,?,-,?,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,?,?,?,-,?,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,?,?,?,?,-,?,?,?,?,?,?,?,?,?,?).
:- block pre(?,?,?,?,?,?,?,-,?,?,?,?,?,?,?,?,?).
% wait until M is instantiated
:- block pre(?,?,?,?,?,?,?,?,-,?,?,?,?,?,?,?,?).
pre(A, B, C, D, E, F, G, H, M, A1, B1, C1, D1, E1, F1, G1, H1) :-
    A1 is A*M+C,
    B1 is C*M+D,
    C1 is A,
    D1 is B,
    E1 is E*M+G,
    F1 is F*M+H,
    G1 is E,
    H1 is F.

% divide top and bottom by gcd i.e. simplify
:- block simplify_rat(-, ?).
simplify_rat(N-D, Result) :-
    simplify_aux(N, D, Result).

% wait for numerator N
:- block simplify_aux(-, ?, ?).
% wait for denominator D
:- block simplify_aux(?, -, ?).
simplify_aux(N, D, N1-D1) :-
    gcd(N, D, X),
    N1 is N/X,
    D1 is D/X.
```

14. APPENDIX V - Sample Output

14.1. Addition

```
| ?- run_this(X).

step           in out  A B C D/ E F G H
-------------- -- ---  ----------------
pre-multiply   1       0 1 1 0/ 0 0 0 1
post-multiply  1       1 1 0 1/ 0 1 0 0
pre-multiply   1       2 1 1 0/ 1 0 0 0
post-multiply  1       3 1 2 1/ 1 0 1 0
pre-multiply   1       4 3 3 2/ 1 1 1 1
post-multiply  1       7 5 4 3/ 2 2 1 1
pre-multiply   1       12 7 7 4/ 4 2 2 1
post-multiply  1       19 11 12 7/ 6 3 4 2
Euclidean step   3     30 19 19 12/ 9 6 6 4
pre-multiply   1       9 6 6 4/ 3 1 1 0
post-multiply  1       15 10 9 6/ 4 1 3 1
pre-multiply   1       25 15 15 9/ 5 4 4 3
post-multiply  1       40 24 25 15/ 9 7 5 4
pre-multiply   1       64 40 40 25/ 16 9 9 5
post-multiply  1       104 65 64 40/ 25 14 16 9
Euclidean step   4     169 104 104 64/ 39 25 25 16
pre-multiply   1       39 25 25 16/ 13 4 4 0
post-multiply  1       64 41 39 25/ 17 4 13 4
pre-multiply   1       105 64 64 39/ 21 17 17 13
post-multiply  1       169 103 105 64/ 38 30 21 17
pre-multiply   1       272 169 169 105/ 68 38 38 21
post-multiply  1       441 274 272 169/ 106 59 68 38
Euclidean step   4     715 441 441 272/ 165 106 106 68
pre-multiply   1       165 106 106 68/ 55 17 17 0
post-multiply  1       271 174 165 106/ 72 17 55 17
pre-multiply   1       445 271 271 165/ 89 72 72 55
post-multiply  1       716 436 445 271/ 161 127 89 72
pre-multiply   1       1152 716 716 445/ 288 161 161 89
post-multiply  1       1868 1161 1152 716/ 449 250 288 161
Euclidean step   4     3029 1868 1868 1152/ 699 449 449 288
pre-multiply   1       699 449 449 288/ 233 72 72 0
post-multiply  1       1148 737 699 449/ 305 72 233 72
pre-multiply   1       1885 1148 1148 699/ 377 305 305 233
post-multiply  1       3033 1847 1885 1148/ 682 538 377 305
pre-multiply   1       4880 3033 3033 1885/ 1220 682 682 377
post-multiply  1       7913 4918 4880 3033/ 1902 1059 1220 682
Euclidean step   4     12831 7913 7913 4880/ 2961 1902 1902 1220
pre-multiply   1       2961 1902 1902 1220/ 987 305 305 0
post-multiply  1       4863 3122 2961 1902/ 1292 305 987 305
pre-multiply   1       7985 4863 4863 2961/ 1597 1292 1292 987
post-multiply  1       12848 7824 7985 4863/ 2889 2279 1597 1292
pre-multiply   1       20672 12848 12848 7985/ 5168 2889 2889 1597
post-multiply  1       33520 20833 20672 12848/ 8057 4486 5168 2889
Euclidean step   4     54353 33520 33520 20672/ 12543 8057 8057 5168
pre-multiply   1       12543 8057 8057 5168/ 4181 1292 1292 0
post-multiply  1       20600 13225 12543 8057/ 5473 1292 4181 1292
pre-multiply   1       33825 20600 20600 12543/ 6765 5473 5473 4181
post-multiply  1       54425 33143 33825 20600/ 12238 9654 6765 5473
pre-multiply   1       87568 54425 54425 33825/ 21892 12238 12238 6765
```

```
  post-multiply   1        141993  88250  87568  54425/  34130  19003  21892
12238
  Euclidean step    4     230243 141993 141993 87568/  53133  34130  34130
21892
  pre-multiply   1      53133 34130 34130 21892/ 17711 5473 5473 0
  post-multiply  1      87263 56022 53133 34130/ 23184 5473 17711 5473
  pre-multiply    1        143285  87263  87263  53133/  28657  23184  23184
17711
  post-multiply   1       230548  140396  143285  87263/  51841  40895  28657
23184
  pre-multiply    1       370944  230548  230548  143285/  92736  51841  51841
28657
  post-multiply  1       601492  373833  370944  230548/  144577  80498  92736
51841
  Euclidean  step     4       975325  601492  601492  370944/  225075  144577
144577 92736
  X = [3,4,4,4,4,4,4,4],
  user:integer_stream(_A,1),
  user:integer_stream(_B,1) ?
  yes
  | ?-
```

$$\left[\overline{1}\right]\oplus\left[\overline{1}\right]=\frac{1+\sqrt{5}}{2}+\frac{1+\sqrt{5}}{2}=1+\sqrt{5}=\left[3,\overline{4}\right]\approx 3.23606797749979$$

This is a periodic continued fraction that begins with 3 and has the recurring pattern 4.
The first 8 partial quotients are:
3/1 = 3
13/4 = 3.25
55/17 = 3.235294117647059
233/72 = 3.236111111111111
987/305 = 3.236065573770492
4181/1292 = 3.236068111455108
7711/5473 = 3.236067970034716
75025/23184 = 3.236067977915804
These were calculated using the Continued Fraction Calculator [19].

14.2. Multiplication

```
| ?- run_this(X).

step           in out   A B C D/ E F G H
-------------- -- ---  ----------------
pre-multiply   1       1 0 0 0/ 0 0 0 1
post-multiply  1       1 0 1 0/ 0 1 0 0
pre-multiply   1       1 1 1 1/ 1 0 0 0
post-multiply  1       2 2 1 1/ 1 0 1 0
pre-multiply   1       4 2 2 1/ 1 1 1 1
post-multiply  1       6 3 4 2/ 2 2 1 1
pre-multiply   1       9 6 6 4/ 4 2 2 1
post-multiply  1       15 10 9 6/ 6 3 4 2
Euclidean step    2    25 15 15 9/ 9 6 6 4
pre-multiply   1       9 6 6 4/ 7 3 3 1
post-multiply  1       15 10 9 6/ 10 4 7 3
Euclidean step    1    25 15 15 9/ 14 10 10 7
pre-multiply   1       14 10 10 7/ 11 5 5 2
```

```
post-multiply  1       24 17 14 10/ 16 7 11 5
Euclidean step   1     41 24 24 14/ 23 16 16 11
pre-multiply   1       23 16 16 11/ 18 8 8 3
post-multiply  1       39 27 23 16/ 26 11 18 8
Euclidean step   1     66 39 39 23/ 37 26 26 18
pre-multiply   1       37 26 26 18/ 29 13 13 5
post-multiply  1       63 44 37 26/ 42 18 29 13
Euclidean step   1     107 63 63 37/ 60 42 42 29
pre-multiply   1       60 42 42 29/ 47 21 21 8
post-multiply  1       102 71 60 42/ 68 29 47 21
Euclidean step   1     173 102 102 60/ 97 68 68 47
pre-multiply   1       97 68 68 47/ 76 34 34 13
post-multiply  1       165 115 97 68/ 110 47 76 34
Euclidean step   1     280 165 165 97/ 157 110 110 76
pre-multiply   1       157 110 110 76/ 123 55 55 21
post-multiply  1       267 186 157 110/ 178 76 123 55
Euclidean step   1     453 267 267 157/ 254 178 178 123
X = [2,1,1,1,1,1,1,1],
user:integer_stream(_A,1),
user:integer_stream(_B,1) ?
yes
| ?-
```

$$\left[\overline{1}\right]\otimes\left[\overline{1}\right]=\frac{1+\sqrt{5}}{2}*\frac{1+\sqrt{5}}{2}=\frac{3+\sqrt{5}}{2}=\left[2,\overline{1}\right]\approx 2.618033988749895$$

This is a periodic continued fraction that begins with 2 and has the recurring pattern 1.
The first 8 partial quotients are:
2/1 = 2
3/1 = 3
5/2 = 2.5
8/3 = 2.6666666666666665
13/5 = 2.6
21/8 = 2.625
34/13 = 2.6153846153846154
55/21 = 2.619047619047619
These were calculated using the Continued Fraction Calculator [19].