# EXACT ARITHMETIC USING A VARIABLE-LENGTH P-ADIC REPRESENTATION

R. Nigel Horspool
School of Computer Science
McGill University
805 Sherbrooke St. West
Montreal, Canada, H3A 2K6

Eric C. R. Hehner
Department of Computer Science
University of Toronto
Toronto, Canada, M5S 1A7

## Summary

The p-adic number system is introduced and developed into a form suitable for performing exact arithmetic in computers. The proposed representation has several desirable attributes: the four standard arithmetic operations have a simple consistent form, the programmer has the ability to choose the precise degree of accuracy in his calculations and the variable-length nature of the representation achieves compact encodings.

## 1. Introduction

The usual arabic number system represents the positive integers as a conceptually infinite sequence of digits

$$\cdots\cdots d_i \cdots\cdots d_3 d_2 d_1 d_0$$

which represents the number

$$\sum_{i=0}^{\infty} d_i \, b^i \quad \text{where b is the base}$$

The rule that leading zeros may be omitted results in a finite representation for every positive integer.

The notation can be modified for use as the internal representation of numbers in computers. A common modification is to truncate the sequence to a fixed number of digits:

$$d_{n-1} \; d_{n-2} \cdots d_2 d_1 d_0$$

and a generalization to the rule that integers modulo $b^n$ are represented by the sequence gives us negative integers. Thus the radix-complement number system is often introduced.

Radix-complement integers are convenient for circuit designers, but they do not satisfy all the requirements of programmers. One major limitation is that only a fixed range of integers are representable, forcing the programmer to program in such a way as to avoid overflow. Furthermore, space is wasted because the smaller integers occur more frequently than larger integers and should receive shorter encodings. A second limitation is that rational numbers are not directly representable.

The restriction to a fixed range of integers can be lifted by permitting variable-length numbers encodings. The variability makes storage management more difficult - the amount of storage initially allocated to a variable may prove to be insufficient as the computation proceeds. Some scheme involving indirection to access numbers with longer encodings is almost essential. However an implementation of variable-length integers has been analyzed[1] and a 31% saving in space-time cost, compared to a fixed length encoding, reported. (The space-time product takes into account not just the decrease in storage used but it also includes increases in processing time due to occasional indirect accesses to numbers and the storage management costs.)

The ability to represent rational numbers can be obtained in many ways. One method is to represent rationals by pairs of integers: a numerator and a denominator. In this form, multiplication and division are reasonably easy, but addition and subtraction are harder and normalization is difficult.[4] Since addition and subtraction are the operations needed most often, a representation where those operations are simpler than multiplication and division would be preferable. Such a representation is the Hensel code[9] which, depending on the length of the code, can represent any rational a/b where a and b are not greated than some integer N. The Hensel code is, in fact, a fixed-length analogue of the representation introduced in this paper.

Of course, the most common computer approximation to the rationals is obtained by inserting a radix point into the sequence of digits. It can be implemented by having a special "point" code that can appear in the sequence or, alternatively, the position of an implied radix point can be indicated by a separate exponent field. In the latter case, we have the usual floating-point number representation. The chief drawback is that only rationals such that the denominator (in lowest terms) divides some power of the number system base can be represented. Hence most arithmetic operations introduce a certain amount of "round-off" error.

In the remainder of this paper, we will describe a variable-length scheme for representing rational numbers. The scheme will permit both exact arithmetic and approximate arithmetic, where the programmer can choose the degree of accuracy. (This is not just the usual choice between single precision and double precision available with many machines.) Our other aim in devising the new scheme is to provide simpler, more consistent,

CH1412-6/78/0000-0010$00.75 (c) 1978 IEEE

arithmetic algorithms than are applicable to conventional number representations.

## 2. P-adic Numbers

P-adic numbers were first devised and their properties explored by a mathematician, Kurt Hensel, at the turn of the century. His results appeared in a sequence of publications that culminated in two books[2,3]. More recently, p-adic numbers have been utilized by Krishnamurthy, who has developed a fixed-length encoding suitable for performing exact arithmetic within a limited range[6,7,8].

The p-adic number system has properties that make it attractive for use in many computer applications. Yet most practicioners are unaware of this system. We will therefore develop the representation in a straightforward manner and at the same time, show how the four standard arithmetic operations and conversions to standard notation may be performed.

We begin with the usual notation for non-negative integers. That is, an infinite sequence of digits $\ldots d_3 d_2 d_1 d_0$ where each $d_i$ represents an integer in the range $0 \le d_i < b$, b is the base of the number system and b is an integer greater than one. From this starting point, the representation for negative integers can be found through performing a subtraction. A subtraction yielding a positive result can be performed by the usual method. We therefore see "what happens" when the result is negative. For example, we can try subtracting one from zero in base 10:

```
  .... 0 0 0 0
- .... 0 0 0 1
= .... 9 9 9 9
```

To subtract the rightmost digits, we must perform a "borrow" - yielding 9 for the rightmost digit of the result. The borrow is usually implemented in hardware by subtracting the next digit of the subtrahend plus one from the next digit of the minuend. This gives us another subtraction of one from zero and so the process repeats. The result is an unending sequence of nines.

Through infinite, the sequence can be represented finitely. We will use an apostrophe to indicate that the digit(s) to its left repeat. Thus zero is 0' and minus-one is 9'. By performing more subtractions, we can construct representations for all the negative integers. Some of these are shown in Table 1. The similarity to the radix-complement system should be apparent. To obtain the radix-complement representation of an integer, we simply truncate the infinite sequence of digits. It should therefore not require verification here that the addition and subtraction algorithms work consistently on our representations of the integers. In fact, the usual radix-complement rule for negating a number works also: complement each digit (replace 0 by 9, 1 by 8 etc., in decimal) and then add one.

The usual multiplication algorithm works unaltered, even on the negative integers. For example, multiplying minus two by minus-three proceeds as follows:

```
          9'8
          9'7
      ...99986
     ...99982
    ...99982
   ...99982
      .
      .
   ...0006
0'6
```

The usual division algorithm cannot be applied however. A division that does not have an integer result would force us to introduce the notion of radix point and an infinite sequence of digits to the right of the point to represent a fractional part. A left-to-right division process is also not (obviously) meaningful for a sequence of digits that extends infinitely far to the left. Consequently, we shall divide from right to left. For example, 1092 divided by 7:

```
          156
0'7 ) 0'1092
        0'42
       0'105     result = 0'156
        0'35
         0'7
```

The result is generated digit-by-digit from right to left. We simply select a digit which when multiplied by the divisor, cancels the next digit (working from right to left) of the dividend. We temporarily limit ourselves to certain divisors that make our division process both possible and unambiguous.

The method works just as simply on negative numbers and even when the result is not an integer. As a simple example, minus-one, 9', divided by 3 yields 3', in decimal. As a more complicated example, minus eighty-five divided by thirty-seven:

```
          7295
0'37 )   9'15
        0'185
        9'730
       0'333
       9'640
        0'74
       9'890     result is 729'5
      0'259
      9'730
```

The calculation halts after this finite number of steps because we obtain a difference that has occurred previously. Recognition of where the earlier difference occurred enables us to insert the apostrophe at the correct position in the answer.

Note that the division process is well defined only when the rightmost digit of the divisor is non-zero and relatively prime to the base. In

decimal, this requires the divisor's final digit to be 1, 3, 7 or 9. The restriction is needed to guarantee the existence and uniqueness of an appropriate multiple of the divisor that cancels the rightmost digit of each successive difference in the division process. We will return to the problem of division later, after extending the representation.

## 3. Extending the P-adic Numbers

The general form of a number in our representation is:

$$d_{n+m}\ d_{n+m-1}\ \cdots\ d_{n+1}\ d_n d_{n-1}\ \cdots\ d_2 d_1 d_0$$

This notation, in fact, describes only a subset of the p-adic numbers. If the digit sequence is not required to repeat then some irrational and imaginary numbers are also representable[5]. However, the simple repetition scheme is more convenient for implementation in computers.

The number represented by the digit sequence, above, can be shown to be

$$\sum_{i=0}^{n} d_i b_i \quad - \sum_{i=n+1}^{n+m} d_i b^i/(b^m-1)$$

where b is the number system base. This formula may be obtained algebraically. We omit the derivation here, but as a simple example consider the meaning of 3' in base 10. With the usual rules, $3' = \sum_{i=0}^{\infty} 3b^i = 3*1/(1-b) = 3/(-9) = -1/3$ . We are, of course, outside the radius of convergence of the summation formula used. However, since we do not actually need to perform the summation - only to assign meanings to digit sequences - there is no difficulty.

Working from the formula above, we find that rational fractions of the form r/s, in lowest terms, are representable if and only if s is coprime to the base, b. (If b and s are coprime, them $b^s-1$ is divisible by s and the expansion of r/s in the required form is clearly possible.) Contrast this with the floating-point number system where a rational r/s, in lowest terms, is representable if all the factors of s are also factors of the base b. Our representation scheme and the floating-point number system are complementary in a genuine sense. Combining the two schemes renders all rationals representable. One means of combining them is to insert a radix point. For example,

```
12'34   ÷ 10 = 12'3.4
12'3.4  ÷ 10 = 12'34
12'34   ÷ 10 = 1.2'34
1.2'34  ÷ 10 = .12'34
.12'34  ÷ 10 = .21'234
```

The last example indicates that an exponent notation is probably more appropriate for a machine representation. Hence the last example could be written as 12'34E-5.

The notation does not, in itself, guarantee a unique representation for each rational. For example, 12'34E-5 can be written as:

```
    12'340E-4
or  21'234E-5
or  1212'34-5
```

We adopt the rule however that the normal form is that representation which has the shortest mantissa. This yields a unique normal form for every rational except zero (which can have an arbitrary exponent value). The normalization process has three parts. First, the repeating part of the number must be "factored" if it consists of a repeating subsequence. Second, trailing zeros should be deleted from the mantissa and the exponent incremented as required. Third, the repeating part must be "rolled" as far to the right as possible, eliminating redundant digits from the non-repeating part of the mantissa. The first part of the normalization process is a non-trivial task when the repeating part of the representation contains a large number of digits.

## 4. Arithmetic Algorithms

Some of the algorithms below terminate only when a repeated state in the calculation is recognized. A method attributed to Richard Brent[6] may be the most convenient way of recognizing repeated states. Denoting the i-th state by $s_i$, the method can be written algorithmically as:

```
for i: = 1,2,4,8, ... do
    for j: = i+1, i+2, i+3, ... 2*i do
        if s_j = s_i then
            stop; {repetition with period j-i}
```

This approach requires that only one previous state be remembered. It does not necessarily discover the first repeated state, but this does not cause our arithmetic algorithms any difficulties if results are normalised.

### 4.1 Addition and Subtraction

The number with the smaller exponent is denormalized by appending zeros to the right of the mantissa, so that both numbers have the same exponent. Then both mantissas are added, digit-by-digit from right to left. The repeating parts are rescanned from right to left as many times as are needed. The addition stops only when a repeated state in the calculation is recognized and the period of the cycle determines the length of the repeating part. The result is a possibly unnormalized number in our representation.

### 4.2 Multiplication

Multiplication can be realized as a repeated addition of simple (one digit) multiples of the multiplicand. In the binary system, this is exactly the usual shift-and-add algorithm. A repeated state can be recognized in exactly the same way as with addition or subtraction. Once again, postnormalization of the result is required.

### 4.3 Division

If the number system has a prime number for

its base, division is exactly analogous to multiplication. The correct multiple of the divisor to use is determined by a simple table look-up on the last digit of the dividend remaining at each step. This multiple of the divisor is subtracted from the dividend and the process repeated. In binary, this method is simply shift-and-subtract. As before, the process repeats until a previous state reoccurs.

When the number system has a composite base some extra labour may be required. In the decimal system, for example, all multiples of two and five must be "cast out" of the divisor's mantissa. Suppose that the divisor's mantissa ends in an even digit. In this case, both the dividend and divisor should be multiplied by five. If the final digit is 5, both the dividend and divisor are multiplied by two. This process can be repeated as required (and any trailing zeros appearing in the mantissa are incorporated into the exponent). Once this preprocessing has been accomplished, the division algorithm given above can proceed.

### 4.4 Sign Determination

Given a mantissa in the general form $d_{n+m}\cdots d_{n+1}{}'d_n\cdots d_1 d_0$, the sign of the number is easily determined.

Assuming that the representation is normalized, then the method is

(a) test whether the mantissa is zero (0').
(b) If non-zero, then test whether there are any digits to the right of the quote symbol. If there are none, the number is negative.
(c) Otherwise, compare $d_{n+m}$ and $d_n$. If $d_{n+m} < d_n$, the number is positive. If $d_{n+m} > d_n$, the number is negative. (Note: $d_{n+m} \neq d_n$ for a normalized number.)

### 4.5 Comparisons

The most easily stated method of comparing two numbers is to compute their difference and check the sign of the difference. A more direct algorithm does exist, but its explanation is too long to warrant inclusion here. We note only that four digits at a time (two from each operand) must take part in the comparison process.

### 4.6 Conversion to and from Right-Repeating Form

In standard decimal notation, any real number is representable as a digit sequence that extends infinitely far to the right of the decimal point. When the number is rational, the decimal expansion is guaranteed to be periodic - hence our use of the term "right-repeating form".

There is a very close relationship between our representation of a number and its representation in right-repeating form. As a striking example, consider -1/7 which in our notation is 142857' and as a decimal expansion is $-0.\overline{142857}$. We can take advantage of this relationship to construct simple algorithms for conversion between the two systems.

(a) Conversion to right-repeating form

Simply subtract the repeating part from the non-repeating part with the leading digits aligned and the repeating part extended indefinitely to the right. For example,

$$\begin{aligned} 12'345 &= 345 - 121.\overline{21} = 223.\overline{78} \\ 123'45 &= 45 - 12.\overline{312} = 32.\overline{687} \\ 43'21 &= 21 - 43.\overline{43} = -22.\overline{43} \end{aligned}$$

(b) Conversion from right-repeating form

The steps given under (a) are simply reversed. For example,

$$\begin{aligned} 2.\overline{34} &= 0'2\overline{34} - 34' = 56'8 \\ 1.2\overline{34} &= 12.\overline{34}\text{E-}1 \\ &= 0'12\text{E-}1 - 34'\text{E-}1 = 65'78\text{E-}1 \end{aligned}$$

### 4.7 Approximation

As arithmetic operators are repeatedly applied in the course of a program's execution, the length of representation of results will tend to grow enormously. Therefore storage costs increase and also the speed of the arithmetic operations will be decreased. This is an inevitable consequence of requiring perfect accuracy for all computations.

Not all users, however, require perfect accuracy for all the calculations in their programs. In an ideal computer system, the user should be free to select his own desired accuracy. Rather than define approximate addition, subtraction, multiplication and division operations, we will introduce a separate approximation operator. Given a number n and a tolerance t, an ideal approximation operator would attempt to find the shortest number that is within the desired range $n \pm nt$. Such an operator is very hard to implement. Instead we propose that the number be converted to right-repeating form and then simple truncated. For example, 12'34.567 is first converted to $22.44\overline{578}$ and then truncated to 0'22.4458, to 0'22.446, or to 0'22.45, etc. If the final truncated result has k digits to the right of the radix point, the error introduced by the approximation is less then $b^{-k}$. The desired value of k is an appropriate parameter for this form of approximation.

## 5. Discussion

In common with all representation schemes, our proposal has both advantages and disadvantages. Amongst its advantages we have:

(a) The four standard arithmetic operations process their operands digit-by-digit from right to left. (This is a property also possessed by Hensel codes[9].)

(b) All rational numbers are representable.

(c) It is a relatively simple matter to provide the programmer with complete control over

the accuracy of the arithmetic operations.

(d) Since the most frequently encountered numbers (e.g., zero and one) receive short encodings, storage is used more efficiently than with fixed-length number representations[1].

(e) The representations are easily converted to the familiar right-repeating form.

Amongst its disadvantages, we cite

(a) The length of the representation does not depend, in a uniform way, on the magnitude and accuracy of the number. Hence the storage requirements of a program could be quite unpredictable.

(b) Not all the operations can easily be implemented as right-to-left algorithms. Comparison, normalization and approximation fall into this category.

| | | | |
|---|---|---|---|
| | 0 : 0' | | |
| 1 : | 0'1 | -1 : | 9' |
| 2 : | 0'2 | -2 : | 9'8 |
| 3 : | 0'3 | -3 : | 9'7 |
| 4 : | 0'4 | -4 : | 9'6 |
| | . | | |
| | . | | |
| | . | | |
| 9 : | 0'9 | -9 : | 9'1 |
| 10 : | 0'10 | -10 : | 9'0 |
| 11 : | 0'11 | -11 : | 9'89 |

Table 1


## Acknowledgements

We give our thanks to Bill McKeeman for his comments and to Don Knuth and a referee for providing additional reference material. We gratefully acknowledge the receipt of financial support from the National Research Council of Canada.


## References


1. E.C.R. Hehner, Computer Design to Minimize Memory Requirements. Computer, vol. 9, no. 8, pp. 65-70, Aug. 1970.
2. K. Hensel, *Theorie der algebraischen Zahlen.* Leipzig-Berlin, 1908.
3. K. Kensel, *Zahlentheorie.* Berlin-Leipzig, 1913.
4. B.K.P. Horn, Rational Arithmetic for Mini-Computers. M.I.T., Cambridge, Mass. (IEEE Repository number R77-247).
5. D.E. Knuth, *The Art of Computer Programming* volume 2, Seminumerical Algorithms, Exercise 31, section 4.1. Addison-Wesley, Reading, Mass. 1969.
6. D.E. Knuth, Personal Communication, 1978.
7. E.V. Krishnamurthy, T. Mahadreva Rao, K. Subramanian, Finite Segment P-adic Number Systems with Applications to Exact Computation. Proc. Indian Academy of Sciences, vol. LXXXI, section A, no. 2, 1975.
8. E.V. Krishnamurthy, T. Mahadreva Rao, K. Subramanian, P-adic Arithmetic Procedures for Exact Matrix Computation. Proc. Indian Academy of Sciences, vol. LXXXII, section A, No. 5, 1975.
9. E.V. Krishnamurthy, Matrix Processors Using P-adic Arithmetic for Exact Linear Computations. IEEE Trans. on Comp., vol. C-26, No. 7, pp. 633-639, July 1977.