Towards a fast and reliable software implementation of SLI-FLP hybrid computer arithmetic

Article · January 2006

2 authors, including:

Peter R. Turner
Clarkson University
78 PUBLICATIONS **566** CITATIONS

SEE PROFILE

Proceedings of the 6th WSEAS Int. Conf. on Systems Theory & Scientific Computation, Elounda, Greece, August 21-23, 2006 (pp101-108)

Towards a Fast and Reliable Software Implementation of SLI-FLP Hybrid Computer Arithmetic

XUNYANG SHEN AND PETER R. TURNER
Department of Mathematics and Computer Science
Clarkson University
Potsdam, NY 13699
U.S.A.

Abstract: - In this paper we describe a C++ implementation of a hybrid system combining SLI (symmetric level-index) arithmetic and FLP (floating-point) arithmetic. The principal motivation for the work to be presented is to promote the use of SLI arithmetic as a practical framework for scientific computing. This hybrid arithmetic is essentially overflow and underflow free, and its implementation has shown the suitability to be used in real life computations.

Key-Words: - symmetric level-index, computer arithmetic, overflow, underflow, hybrid, number representation.

1 Introduction

This paper is concerned with the implementation, which could be used in practical computations, of an over/underflow free arithmetic based on *symmetric level-index* (SLI) system. The implementation currently can be run in a C++ environment of either WINDOWS or LINUX, and is available for download at http://www.sliarithmetic.net.

There are other software implementations of SLI arithmetic, typically in MATLAB, FORTRAN [6], or PASCAL [9]. The purposes of these implementations are to verify the algorithms, and to show the effectiveness of a potential hardware implementation by some numerical experiments. Besides those motivations, the new C++ implementation emphasizes more its practical use as software, with some recent improvements of the algorithms and the design of a hybrid number representation scheme. The choice of programming language is based on the fact that C++ is efficient, flexible, versatile, and popular in scientific computation.

We claim the implementation is appropriate for use in some real life computational problems, where over/underflow causes trouble. A systematic testing process was executed to show its reliability and efficiency in section 4. An application is also described in [8] with some performance records of the program.

Ease of use is also an important concern for the implementation. Little knowledge about SLI arithmetic is required, and no additional analysis to the computational problem is needed to use the C++ library. In other words, it is not necessary to predict which variable has the risk of overflow or underflow in floating-point (FLP) arithmetic, or to estimate how large or how small the result could be, if the FLP representable range is to be exceeded. When the desired operations in a program are supported by this C++ implementation, all a programmer needs to do is to simply replace the data type *double* with type *SLI* for all real number variables.

The majority of this paper will discuss the design of the hybrid number representation scheme, and the algorithms to perform arithmetic operations, as well as a few implementational details of them in C++. A little overview of SLI arithmetic is provided meanwhile.

2 Hybrid Number Representation

The *level-index* (LI) number representation is to represent a positive real number X by x, where

$$X = \phi(x) = \begin{cases} x, & 0 \le x < 1, \\ e^{\phi(x-1)}, & x \ge 1, \end{cases} \tag{1}$$

and ϕ is called the *generalized exponential function*. It follows that $x = l + f$, where the *index* $f \in [0,1)$ is given by

$$f = \ln(\ln(...(\ln X))), \tag{2}$$

the natural logarithm being taken l times. So the *level* l must be a nonnegative integer.

The symmetric level-index system presents a number in (0, 1) by the LI image of its reciprocal. Thus a real number X can be represented by

$$X = \pm\phi(x)^{\pm 1} = s_X \phi(l_X + f_X)^{r_X}, \tag{3}$$

and four pieces of information are needed to store a SLI number -- number sign s_X, level l_X, index f_X, and reciprocal sign r_X.

SLI number representation has an essentially infinite representable number range. It is sufficient to allocate just 3 bits to the level when working to any conceivable implemented precision, and the four basic arithmetic operations would be closed in the finite number set of SLI representation.

The details of LI and SLI number representation and their algorithms for arithmetic operations are described in [2], [4], [3], [1], and [7]. It is not surprising that to perform arithmetic operations would be more complicated in the SLI system than in FLP arithmetic. Accordingly a hybrid system is proposed in [8] to take the advantages of both SLI arithmetic and FLP arithmetic. If the magnitude of a number is within a certain range, $[2^{-511}, 2^{511}]$ for example, the number is represented in the IEEE standard form [5]. SLI number representation is used otherwise.

In this hybrid system, a double precision number can be packed into a 64-bit word. See [8] for a possible representation scheme in a potential hardware implementation. However, it is not efficient to program this scheme in C++. We implemented the hybrid number representation using a structure of total length of 70 bits, as illustrated in Fig. 1. Note a SLI number in this hybrid system is at least level 4, since $\phi(4.571) \approx 2^{511}$

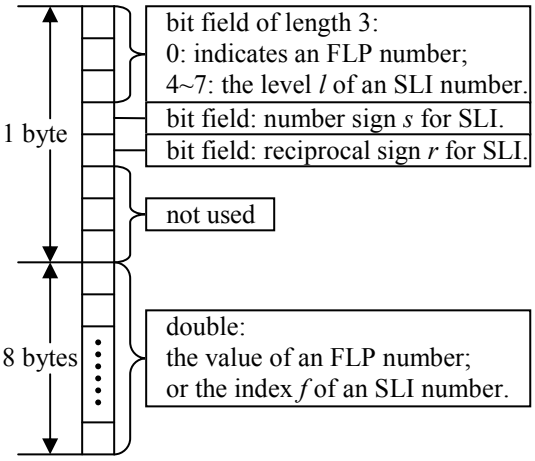

Fig. 1: SLI-FLP hybrid number representation implemented in C++

This way no bit manipulations are required, and the software implementation can run faster, with the tradeoff of a little extra space complexity.

3 Algorithms for Arithmetic Operations

The algorithms implemented are based on the modified algorithms for SLI arithmetic operations described in [1]. Those modifications allow more parallel implementations in a potential hardware design. With a small change of initial calculations which will be described later, the modified algorithms also benefit a software implementation.

There are a few major modifications to the algorithms in [1], including the treatments of hybrid number representation, and an approximation method based on Taylor expansions for the algorithm of addition/subtraction.

3.1 Algorithm for Addition/Subtraction

The addition/subtraction problem is to solve the equation

$$Z = s_Z \phi(z)^{r_Z} = s_X \phi(x)^{r_X} \pm s_Y \phi(y)^{r_Y} = X \pm Y \quad (4)$$

for s_Z, r_Z, z given s_X, s_Y, r_X, r_Y, x, y. We may assume $\phi(x)^{r_X} \geq \phi(y)^{r_Y}$ (or $|X| \geq |Y|$) without losing generality.

In the hybrid system, different paths of the algorithm will be executed for operands in different ranges. The figure below displays which path would be chosen according to the larger magnitude operand in the addition/subtraction algorithm.

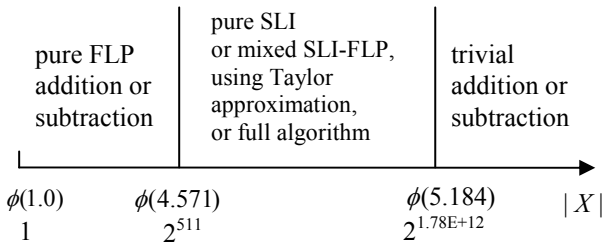

Fig. 2: Domain decomposition of the algorithm for addition/subtraction in the hybrid system, when the larger magnitude operand X satisfies $|X| > 1$

The domain decomposition is almost symmetric about $\phi(1.0)$ noticing $\phi(1.0) = 1 = \phi(1.0)^{-1}$. So it is only necessary to show the number line for $|X| > 1$. Both SLI and FLP numbers are used to denote magnitudes in the number line. Notice a double precision FLP number can only reach 2^{1024}, which is about $\phi(4.632)$. The algorithm path in the second subdomain is most complex, and can not be shown in a single illustration. Figures of finer domain decomposition can be found in [7] and [8].

Trivial arithmetic operations are defined as the cases where the operation result equals to the larger magnitude operand. In double precision arithmetic, addition/subtractions are all trivial except for $\phi(x) - \phi(x)$ if $\phi(x) \geq \phi(5.18393800805500)$, so $Z = X$ and the algorithm is complete.

The most beneficial modification in the algorithm is to adopt FLP arithmetic operations. A pure FLP addition or subtraction will be performed, if both the

operands are within the reduced FLP range [2^{-511}, 2^{511}]. It will be much faster than a pure SLI operation in any conceivable software implementations. If the larger magnitude operand is a FLP number and the other operand is a SLI number, it is safe to convert the SLI operand into FLP form and do a pure FLP operation. But for the other SLI-FLP mixed case, the SLI algorithm needs to be performed.

The modified full algorithms are based on computing sequences defined by

$$a_j = 1/\phi(x-j);$$
$$b_j = 1/\phi(y-j); \quad (5)$$
$$c_j = \phi(z-j)/\phi(x-j).$$

Assuming $s_X = s_Y = 1$ in equation (4) for simplicity, we give the full algorithm for SLI addition or subtraction, which is modified from [1] as follows.

Algorithm 1: Modified full SLI addition/subtraction of positive arguments for nontrivial cases

Input (r_X, x), (r_Y, y)

where $\phi(x)^{r_X} \geq \phi(y)^{r_Y}$ and $l_X, l_Y \geq 2$.

$$large = (r_X == r_Y == 1);$$

Initialize Booleans $mixed = (r_X != r_Y)$;

$$small = (r_X == r_Y == -1).$$

$r_Z = r_X$.

Compute $a_{l_X - 1} = \exp(-f_X)$,

$a_{j-1} = \exp(-1/a_j)\ (j = l_X - 1,...,2)$;

$b_{l_Y - 1} = \exp(-f_Y)$,

$b_{j-1} = \exp(-1/b_j)\ (j = l_Y - 1,...,2)$;

$$d = \begin{cases} \exp\left(\dfrac{a_1 - b_1}{a_1 b_1}\right), & large \\ \exp\left(-\dfrac{a_1 + b_1}{a_1 b_1}\right), & mixed \\ \exp\left(\dfrac{b_1 - a_1}{a_1 b_1}\right), & small \end{cases}$$

$c = 1 \pm d$

$j = 1$

$c = 1 + r_X a_1 \ln c$

while $j < l_X - 1$ and $c \geq a_j$

$j = j + 1, c = 1 + a_j \ln c$

if $c < a_j$ then $z = j + c/a_j$, go to **Output**

$j = l_X, h = f_X + \ln c$

while $h \geq 1$

$j = j + 1, h = \ln h$

$z = j + h$

Output (r_Z, z)

Operator "$\pm$" is denoted to compute the initial value of the *c*-sequence. It means that "+" is used for addition and "–" is used for subtraction.

The computation of sequences in this algorithm is mathematically but not numerically equivalent to what is defined in [1]. In order to avoid possible overflow/underflow problems when generating the *c*-sequence with FLP internal operations, the initial value of the *c*-sequence is calculated in a different way. The new definition of the *c*-sequence is also more efficient to compute in a software implementation, by saving one exponential function evaluation. The tradeoff is that it requires both operands to be at least level 2. But this will not be a problem in the hybrid system, since then all SLI numbers start from level 4.

Another saving is that the *flipover* cases are all avoided. A flipover addition is defined as the case where the reciprocal sign of the larger addend is changed after a SLI addition. But any two SLI numbers with different reciprocal signs are sufficiently distant in this hybrid system to avoid flipover additions, considering the limited precision of a 64-bit number representation.

Algorithm 1 is the full algorithm for pure SLI addition or subtractions. As shown in Fig. 2, the algorithm will be considered when the larger magnitude operand X satisfies $X = s_x \phi(x)^{r_X}$ and $2^{511} < \phi(x) < \phi(5.184)$. There can be two simple variations for the algorithm, depending on the magnitude of the other operand *Y*.

When *Y* is represented as a FLP number in this hybrid system, it is not necessary to compute the *b*-sequence. Notice only b_1, the last value in the sequence, is useful in the later computations. It is possible to skip generating the *b*-sequence in the SLI-FLP mixed case, by redefining

$$d = Y \cdot \exp(-1/a_1) \quad (6)$$

and performing a large addition in the rest of the algorithm. This variation can be easily verified according to the sequence definitions in (5).

When *Y* has a magnitude sufficiently smaller than *X*, the Taylor approximation scheme (see [7] for details) is applicable and the computation of the *c*-sequence can be avoided. In this particular implementation, a linear interpolation is performed for each addition or subtraction to decide if the first order Taylor approximation method is useful. If it is, calculate

$$\varepsilon = a_1 a_2 ... a_{l_X - 1} \cdot \begin{cases} d, & large \text{ or } mixed \\ d/(d+1), & small \end{cases} \quad (7)$$

and output $(r_X, x \pm \varepsilon)$. The approximation of ε is based on a different expression than in [7], because the original full algorithm is used in [7], while the modified algorithm is adapted in this paper. However, the formula deduction of ε is very similar.

3.2 Error Bounds in Implementation

Since all the internal operations are performed in standard double precision FLP arithmetic, the error of the implemented SLI arithmetic operations is larger than what is predicted similarly as in [4].

When performing an SLI addition, we assume that the sequence $\{a_j\}$ is stored with absolute precision γ while all the other quantities are stored to absolute precision γ_1. Let δz represent the error of z caused by inherent errors δx in x and δy in y. As stated in [4], δz in a large addition satisfies

$$\begin{aligned} |\delta z| &\le |c_{l_z - 1}| + \gamma \\ &\le (2\rho + 2)\gamma + (1/e - \ln\gamma)\rho\lambda\gamma_1 \end{aligned} \quad (8)$$

where

$$\rho = 1 + 1/\phi'(1) + 1/\phi'(2) + ... \approx 2.3921 \quad (9)$$

and

$$\lambda = (4 + e)e^{-e} + 1 \approx 1.7515 \quad (10)$$

On replacing ρ, λ and e by their numerical values, the result reduces to

$$|\delta z| \le 6.784\gamma + (1.541 - 4.190\ln\gamma)\gamma_1 \quad (11)$$

Due to the restriction of FLP arithmetic, the internal operations in this software implementation have the precision of 2^{-52}. Substituting both γ and γ_1 with 2^{-52} in (11), we computed the error $|\delta z|$ in a large addition is bounded by approximately 3.65×10^{-14}, and similar error bounds can be obtained for mixed and small cases. The largest among the error bounds is 3.65×10^{-14}, which would be acceptable for most computations.

Although the above error analysis is based on the full algorithm for pure SLI addition, the error bounds also work for the variations of the algorithm. The SLI-FLP mixed case is expected to have a better precision since the roundoff error in generating the *b*-sequence is avoided. The Taylor approximation is deemed to be applicable only when its result perfectly matches the result of the full algorithm at the expected precision level, so the approximation method does not introduce any further error to the implementation.

Subtractions in this implementation have similar errors, unless cancellations occur. The correctness testing in section 4 supports the error analysis in this section.

3.3 Algorithm for Multiplication/Division

The domain decomposition of multiplication/division is similar to the one for addition/subtraction, except that trivial cases start from a different number. The threshold is about $\phi(6.1839)$ for multiplication/ division in double precision SLI arithmetic.

The full algorithm for multiplication or division is also modified from [1]. But note here $\phi(x) \ge \phi(y)$ instead of $\phi(x)^{r_X} \ge \phi(y)^{r_Y}$ is required for a pure SLI multiplication/division.

Algorithm 2: Modified full SLI multiplication/ division of positive arguments for nontrivial cases

Input (r_X, x), (r_Y, y)
where $\phi(x) \ge \phi(y)$ and $l_X, l_Y \ge 3$.

Initialize Booleans *large*, *mixed*, *small*,
$r_Z = r_X$.

Compute $a_{l_X - 1} = \exp(-f_X)$,
$a_{j-1} = \exp(-1/a_j)\ (j = l_X - 1, ..., 3)$;
$b_{l_Y - 1} = \exp(-f_Y)$,
$b_{j-1} = \exp(-1/b_j)\ (j = l_Y - 1, ..., 3)$;

$$c_1 = \begin{cases} 1 \pm \exp\left(\dfrac{a_2 - b_2}{a_2 b_2}\right), & large \text{ or } small \\ 1 \mp \exp\left(\dfrac{a_2 - b_2}{a_2 b_2}\right), & mixed \end{cases}$$

$c = 1 \pm a_2 \ln c_1$
$j = 2$
while $j < l_X - 1$ and $c \ge a_j$
  $j = j + 1, c = 1 + a_j \ln c$
if $c < a_j$ then
  if $j = 2$ and $c_1 < \exp(-1/a_2)$ then
    $z = 1 + c_1 / \exp(-1/a_2)$, go to **Output**
  else $z = j + c/a_j$, go to **Output**
$j = l_X, h = f_X + \ln c$
while $h \ge 1$
  $j = j + 1, h = \ln h$
$z = j + h$

Output (r_Z, z)

When a "$\pm$" operation appears in this algorithm, "$+$" should be used for multiplication and "$-$" would be used for division.

This algorithm can also have variations to make it run faster for some special cases. However, the Taylor approximation method is not very helpful for multiplication/division, since its applicable domain is very limited.

The SLI-FLP mixed case can still simplify the problem. We first swap X with Y if the leading operand X is an FLP number. Then the value of c_1 can be computed by

$$c_1 = \begin{cases} 1 \pm \exp(-1/a_2) \cdot \ln|Y|, & large \text{ or } mixed \\ 1 \mp \exp(-1/a_2) \cdot \ln|Y|, & small \end{cases} \quad (12)$$

Again the b-sequence is not needed.

The multiplication/division algorithm is expected to have similar error as the addition algorithm, unless cancellations happen.

3.4 Algorithm for Powering Operation

It is possible to derive a similar algorithm for powering operation, but there would be many special cases to consider. So the powering operation was implemented by calling the subroutine of multiplication. For example, if

$$\phi(z) = \phi(x)^{\phi(y)}, \quad (13)$$

taking logarithms of both sides of the equation, we will get

$$\phi(z-1) = \phi(x-1)\phi(y). \quad (14)$$

So the large case of powering problem becomes a multiplication problem at different levels. However, a separate powering algorithm could be marginally faster because of the saving of extra function calls.

4 Testing

One important purpose of this software implementation is to promote the use of SLI arithmetic in practical scientific computation, so the correctness and the performance of the program are both critical. A systematic testing procedure was designed and executed, so that we can ensure the usefulness of the implementation by providing the testing results.

4.1 Correctness Testing

We are not able to directly test the correctness of this SLI-FLP hybrid system. It is easy to verify the cases where FLP arithmetic operations are performed, but it will be hard to tell if the results are correct for numbers that exceed the double precision FLP representable range, especially for cases using the Taylor approximation method.

So our testing began from the pure SLI system using only the full original algorithm introduced in [4]. The original algorithm is identical to the modified algorithm in [1] except for the definitions of the b-sequence. But there are fewer restrictions that are needed for a pure SLI system using FLP internal operations.

We may compare the results from pure SLI arithmetic with the ones from FLP arithmetic if the magnitudes of testing data are bounded, so that the operations will not cause overflow or underflow in FLP arithmetic. A large number of such tests are able to cover all branches of the algorithm. If the implementation gives correct results for all tests in the FLP range, we may reason that the algorithm remains correct for numbers that exceed this range, because all cases in the pure SLI system are essentially the same computation at different levels.

Next we can test the hybrid arithmetic based on the pure SLI system. The agreement of their results in arithmetic operations is enough to conclude the correctness of the implementation of the hybrid system.

To test the pure SLI arithmetic using only the full original algorithm, the testing cases were chosen both by hand and by a random number generator.

10,000 pairs of random operands with a log-normal distribution in magnitudes, as shown in Fig. 3, were generated for the testing of the four basic arithmetic operations. The log-normal distribution is the best choice for the testing cases among the well known distributions, because the testing needs to cover a range of double precision FLP numbers as large as possible, but at the same time, sufficient testing cases need to be picked around 1, in order to test all the three flipover cases which appear infrequently.

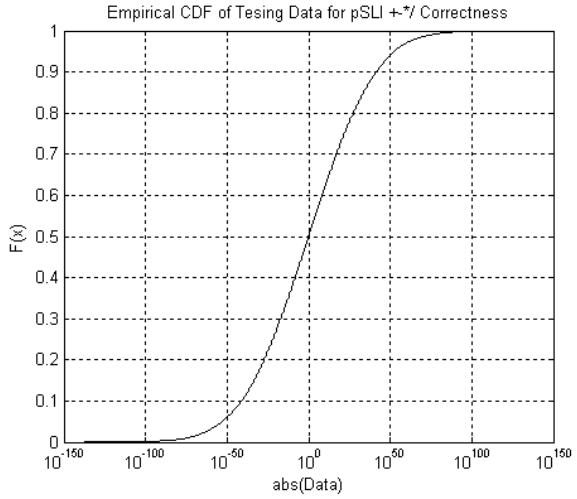

Fig. 3: Cumulative distribution of the correctness testing data for basic arithmetic operations in the pure SLI system

The random operands for testing the powering operation could be generated similarly. But FLP overflow/underflow happens more frequently in powering than in the basic arithmetic operations, so the number range of the testable cases is very limited, and a normal distribution is used instead.

A testing tool in LINUX called GCOV is used to verify the coverage of all branches of the algorithm. When some cases with a true zero in operations are manually added, every line in the arithmetic operation subroutines is covered in testing.

We claim a test successful if the result in SLI arithmetic and the result in FLP arithmetic differ by no more than a small tolerance. According to the error bounds predicted in Section 3.2, a tolerance of 3.65×10^{-14} is used in the index. The FLP result of an operation is converted into an SLI number, and its index is compared with the index of the SLI result, in the situation that both results have the same level in SLI forms.

Using the tolerance 3.65×10^{-14}, all tests of addition/subtraction were successful. A few tests would fail if a smaller tolerance such as 2×10^{-14} is used. The results verified the error bounds predicted in section 2.

Almost all the tests of multiplication/division and powering operation succeeded with the same error tolerance. A few exceptions were observed, since cancellations occur more often in multiplications and divisions, with the log-normal distributed testing data.

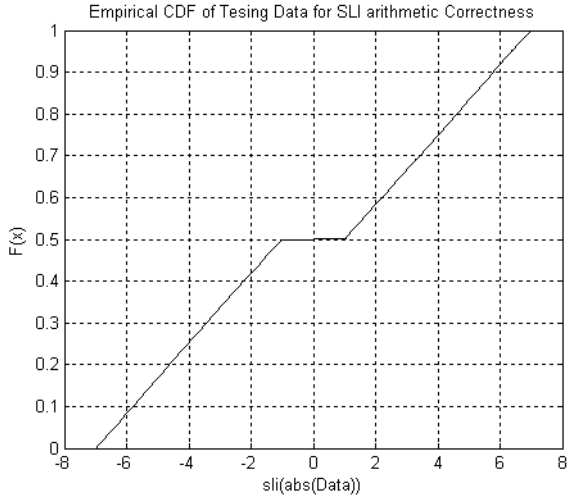

Fig. 4: Cumulative distribution of the correctness testing data for arithmetic operations in the SLI-FLP hybrid system

Next we tested the implementation of the hybrid system. The random number generator can now work with SLI numbers to get a proper distribution of testing data, and the uniform distribution, as shown in Fig. 4, works well for the testing of all arithmetic operations, including powering. Note the signs of the horizontal coordinates denote for the reciprocal signs of SLI numbers instead of number signs. The reason for having a horizontal line segment in the center is that $\phi(1.0) = 1/\phi(1.0)$, and the length of this line segment is zero in SLI arithmetic.

For the hybrid system, we have tested 10,000 pairs of random operands following a uniform distribution in magnitude, plus a few special cases manually added. All branches of the algorithm were tested, and the results matched the ones from FLP arithmetic or from the well tested pure SLI system. Hence we conclude that to perform arithmetic operations in this SLI-FLP hybrid system is reliable.

4.2 Performance Testing

In order to evaluate the performance, the time costs of performing arithmetic operations in the SLI implementation will be compared to the costs in FLP arithmetic. However, the SLI arithmetic was programmed in C++ as a class, and has a lot of overhead in running time, while the FLP arithmetic was built in chips. The results of a direct comparison will not reflect the efficiency difference between their algorithms.

An FLP class was then programmed to make the comparison fairer. In this performance comparison we still run the functions defined by ANSI C++ to do FLP arithmetic operations, but all the operations are now encapsulated into a class, in order to add similar overhead to FLP computations as the ones SLI arithmetic has in the software implementation. Note the comparison is not yet fair for SLI arithmetic, since the logical structure of the algorithm cannot be efficiently programmed in a high level language.

The performance testing was done for four types of arithmetic operations -- addition/subtraction, multiplication, division and powering. 10,000 random numbers were generated as operands, and each operation between any two numbers was repeated thousands of times for a more accurate comparison. Because the hybrid system uses different algorithms to perform arithmetic operations for operands in different number ranges, we made three tests with different sets of testing data. All three sets of data follow a uniform distribution in SLI representation, but with various number ranges as shown in Table 1.

The number range specified in the table is only for those greater or equal to 1. If a number X is between 0 and 1, it must be within the reciprocal of the corresponding range. For example, X must be within $(2^{-1022},1)$ in data set 1. There are negative numbers

Table 1: Properties of data sets used in performance testing

Data set	Number range (for those ≥ 1.0)	Operands in FLP over/underflow	Results in FLP over/underflow
1	$(1,2^{1022})$	0%	3.31%
2	$(\phi(4.6322),\phi(7.0))$	100%	100%
3	$(1,\phi(7.0))$	39.09%	50.05%

as well, and their absolute values fall in the same number ranges.

Data set 1 contains only numbers in the double precision FLP representable range. However, it is possible that the results of arithmetic operations overflow or underflow in FLP arithmetic, especially in powering operations. The performance of the implementation in this number range is our major concern, since most practical computations are still in this range. The time costs of other operations in different systems are shown in the following table, providing that an FLP addition/subtraction costs one time unit. The experiments were performed using a desktop with a 3GHz Pentium 4 CPU.

Table 2: Time comparison for operands in double precision FLP range, normalized with respect to FLP addition

Operation	FLP	SLI	hybrid
X +(-) Y	1	72.2517	5.19868
X * Y	1.05132	73.1457	6.71523
X / Y	2.06625	73.2781	6.80794
X^Y	15.6821	90.7615	31.0729

The pure SLI system used in comparison is implemented based on the original algorithm without the Taylor approximation method. The results in Table 2 clearly show that the hybrid system benefits a lot from using FLP arithmetic in an appropriate number range. The hybrid system costs less than five times the FLP arithmetic does. The cost ratio will be even better in practice, since a real life problem usually spends a large portion of time on operations other than real number computations. See [8] for an example.

Table 3: Time comparison for operands beyond FLP range, normalized with respect to FLP addition

Operation	FLP	SLI	hybrid
X +(-) Y	3.93377	43.0465	29.8079
X * Y	3.96689	60.9270	42.3445
X / Y	5.15232	60.7948	47.0198
X^Y	5.67219	103.047	97.0726

Data set 2 contains numbers beyond double precision FLP range. In this number range we may tell how the Taylor approximation scheme and the other modifications to the algorithm benefit the performance of SLI arithmetic operations. The time comparisons are displayed in Table 3.

Note that the FLP results are all exceptions (0, Inf or NaN), so the times indicated do not reflect meaningful results.

Data set 3 is used to evaluate the overall performance of the hybrid system, so both FLP numbers and SLI numbers are included.

Table 4: Time comparison for operands uniformly distributed in the hybrid number representable range, normalized with respect to FLP addition

Operation	FLP	SLI	hybrid
X +(-) Y	2.41722	65.7285	25.1193
X * Y	2.43212	71.3907	34.9537
X / Y	3.58609	71.4239	38.5164
X^Y	10.4636	97.1193	65.2583

In Table 4, the hybrid system clearly shows its advantage over the pure SLI system without the Taylor approximation scheme. The overall performance is almost doubled for most operations. Although the FLP arithmetic is still much faster, it does not always give desirable results. As shown in Table 1, 39% of numbers and 50% of operations overflow or underflow in FLP arithmetic for this data set.

The performance testing shows that the hybrid system works much faster than a pure SLI system, and its performance is acceptable even compared with a pure FLP system, when a comparison is appropriate.

5 Conclusion

The software implementation of SLI-FLP hybrid arithmetic described in this paper is robust and ready to be used in practical computations. The number representation scheme and the algorithms for arithmetic operations were all modified, with the goal of being efficiently implemented using a high level computer language. In order to ensure the reliability and efficiency of the implementation, both correctness testing and performance testing were carefully designed and executed.

With acceptable efficiency, this C++ library can expediently help mathematicians, engineers and

programmers to control overflow and underflow problems in computing. We hope the implementation also help to motivate further research on SLI arithmetic, by bringing it to the interest of more people who need its advantages.

References:

[1] M.A. Anuta, D.W. Lozier, N. Schabanel & P.R. Turner, Basic Linear Algebra Operations in SLI Arithmetic, *Computing and Applied Mathematics Laboratory (Applied and Computational Mathematics Division), NIST Report,* NISTIR5811, 1996.

[2] C.W. Clenshaw & F.W.J. Olver, Level-index Arithmetic Operations, *SIAM J. Numer. Anal.* 24, 1987, 470-485.

[3] C.W. Clenshaw, F.W.J. Olver & P. R. Turner, *Level-Index Arithmetic: An Introductory Survey, Numerical Analysis and Parallel Processing (P.R. Turner, ed.)*, Springer-Verlag, Berlin, 1989, 95-168.

[4] C.W. Clenshaw & P.R. Turner, The Symmetric Level-Index System, *IMA J. Numer. Anal.* 8, 1988, 517-526.

[5] IEEE 754/854, *IEEE Standard for Floating Point Arithmetic*, ANSI/IEEE Std. Vols 754/854, IEEE, New York.

[6] D.W. Lozier & P.R. Turner, Symmetric Level Index Arithmetic in Simulation and Modeling, *J. Res. Natl. Inst. Stand. Technol.* 97, 1992, 471-485.

[7] X. Shen & P.R. Turner, Taylor Approximation for Symmetric Level-Index Arithmetic Processing, *IMA J. Numer. Anal.*26, 2006, 584-603.

[8] X. Shen & P.R. Turner, A Hybrid Number Representation Scheme Based on Symmetric Level-Index Arithmetic, *CSC'06 (WORLDCOMP'06)*, CSREA Press, Las Vegas, 2006, 118-123.

[9] P.R. Turner, A Software Implementation of SLI Arithmetic, *Proc. ARITH9*, IEEE Computer Society, Washington DC, 1989, 18-24.