# Complex SLI Arithmetic: Representation, Algorithms and Analysis

Peter R Turner
Mathematics Department, US Naval Academy, Annapolis, MD 21402

**Abstract**
*In this paper the extension of the SLI system to complex numbers and arithmetic is discussed. The natural form for representation of complex quantities in SLI is in the modulus-argument form and this can be sensibly packed into a single 64-bit word for the equivalent of the 32-bit real SLI representation. The arithmetic algorithms prove to be very little more complicated than for real SLI arithmetic. The paper describes the representation, arithmetic algorithms and the control of errors within these algorithms.*

**Key Words** Symmetric Level-Index, SLI, complex arithmetic, representation, algorithms, error analysis

## 1. Introduction

The level-index number system for computer arithmetic was first suggested in Clenshaw and Olver [1],[2]. The scheme was extended to the symmetric level index, SLI, representation in [4] and has been studied in several further papers in the last few years. Much of the earlier work is summarized in the introductory survey [3]. The primary virtue of SLI arithmetic is its freedom from overflow and underflow and the consequent ease of algorithm development available to the scientific software designer. This is not the only arithmetic system that has been proposed with this aim; for example the work of Matsui and Iri [10], Hamada [6], [7] and Yokoo [18] suggested and studied modified floating-point systems which share some of the properties of level-index.

Possible hardware implementations of SLI arithmetic were discussed in [13], [14], [17] while a software implementation incorporating some extended arithmetic was described in [16]. The error analysis of SLI arithmetic is discussed in [2], [4] and is extended in [8], [11], [12]. Applications and software engineering aspects of the level-index system have been discussed in [5],[9], [15].

In this paper, we concentrate on the use of the SLI representation and arithmetic for scientific computing using complex arithmetic. In many complex scientific computing situations, such as the Fast Fourier Transform, the computation would be significantly facilitated by having complex variables represented and processed in their modulus-argument form. This polar form turns out to be the natural form of representation for SLI complex arithmetic and the arithmetic algorithms using this representation are not greatly more complicated than for real SLI arithmetic.

In Section 2, we describe the representation of complex variables within the SLI system. The representation adopted uses a single 64-bit word for the equivalent of the 32-bit real SLI representation. This word consists of two parts which represent, respectively, the argument stored as a fixed-point two's complement fraction of $\pi$ and the standard SLI representation of the modulus.

Section 3 is concerned with the arithmetic algorithms for this representation. Not surprisingly, multiplication and division turn out to be no more complicated than their real counterparts. More interestingly, though, it also turns out that addition and subtraction can be readily achieved in this format. The algorithms needed are only slight complications of the real SLI algorithms. In Section 4, we present an error analysis of complex SLI operations.

We begin with a brief review of the SLI representation and its arithmetic. The symmetric level index representation of a real number $X$ is given by

$$X = s_X \phi(x)^{r_X} \tag{1}$$

where the two signs $s_X$ and $r_X$ are $\pm1$, the *generalized exponential* function is defined for $x \geq 0$ by

$$\phi(x) = \begin{cases} x & 0 \leq x < 1, \\ \exp(\phi(x-1)) & x > 1 \end{cases} \tag{2}$$

and the representation is normalized by the requirement $x \geq 1$. It follows that for $X > 1$,

$$X = \exp(\exp(\dots(\exp f)\dots)) \tag{3}$$

where the exponentiation is performed $l = [x]$ times and $x = l + f$. The integer part, $l$ of $x$ is called the *level* and the fractional part, $f$ is called the *index*. The freedom of this system from over- and underflow results from the fact that, working to a precision of no more than 5,500,000 binary places in the index, the system is closed under the four basic arithmetic operations apart from division by zero. This is discussed briefly in [1], [4] and considered in some detail in [8].

The appropriate error measure for computation in the level index system is no longer relative error (which corresponds approximately to absolute precision in the mantissa of floating-point numbers) but *generalized precision* which corresponds to absolute precision in the index. This error measure is introduced in [1].

For clarity in the description of the complex SLI algorithms in Section 3, it is desirable to review some of the fundamentals of the algorithms of real SLI arithmetic. The algorithms are described in some detail in [4].

The basic problem is that of finding the SLI representation $s_Z\phi(z)^{r_Z}$ of $Z=X\pm Y$ where $X$, $Y$ are also given by their SLI representations. Without loss of generality, we may assume that $X\geq|Y|>0$ so that $s_Z=+1$. The computation entails the calculation of the members of three short sequences which vary according to the particular circumstances. In every case, the sequence defined by

$$a_j = \frac{1}{\phi(x-j)} \qquad (j = l-1, l-2, \ldots, 0) \tag{4}$$

where $l=[x]$, is computed using the recurrence relation

$$a_{j-1} = \exp(-1/a_j); \qquad a_{l-1} = e^{-f}. \tag{5}$$

Depending on the values of $r_X$, $r_Y$ and $r_Z$, the other sequences that may be required are given, for appropriate starting values also determined from $x$, $y$, by

$$b_j = \frac{\phi(y-j)}{\phi(x-j)},\ \beta_j = \frac{\phi(x-j)}{\phi(y-j)},\ \alpha_j = \frac{1}{\phi(y-j)} \\ c_j = \frac{\phi(z-j)}{\phi(x-j)},\ h_j = \phi(z-j). \tag{6}$$

Specifically, the sequence $(b_j)$ is used for *large* arithmetic when $r_Y = +1$. The terms can be computed using

$$b_{j-1} = \exp\left(\frac{b_j - 1}{a_j}\right) \tag{7}$$

with the initial value given by

$$b_{m-1} = a_{m-1}e^{g} = \begin{cases} \exp(g - 1/a_m) & (m < l) \\ \exp(g - f) & (m = l) \end{cases} \tag{8}$$

where $m = [y]$ is the level of $y$. Since, in this case, $y \leq x$, it follows that $0 \leq b_j \leq 1$.

The sequence $(\alpha_j)$ is used for *mixed* arithmetic when $r_X = +1$, $r_Y = -1$ and is computed like $(a_j)$. It is similarly bounded: $0 \leq a_j, \alpha_j < 1$.

For *small* arithmetic where $r_X = r_Y = -1$, the requirement $X \geq Y$ implies $x \leq y$. The sequence $(a_j)$ is computed as before along with the sequence $(\beta_j)$. This latter is computed using the recurrence relation

$$\beta_{j-1} = \exp\left(\frac{\beta_j - 1}{a_j\beta_j}\right) \tag{9}$$

for $j < l$ with the initial value

$$\beta_{l-1} = \begin{cases} \exp(f - 1/\alpha_l) & (l < m) \\ \exp(f - g) & (l = m) \end{cases}$$

The starting values and relations for the $c$- and $h$-sequences are discussed in the context of the extended algorithm in the next section. Like the above sequences, their values are always suitably bounded. See [4] for a detailed discussion of the algorithm, [13], [14] and [17] for possible schemes for hardware implementation.

## 2. Complex SLI representation

In this section we describe the SLI representation of complex quantities. It is consistent with the philosophy of the SLI representation of real numbers that complex numbers should be represented in their modulus-argument - or polar - form. That is, a complex number $Z$ is represented by

$$Z = Re^{i\theta} = \phi(r)^{\pm 1}e^{i\theta} \tag{10}$$

This is equivalent to writing Z as the exponential of another complex number (its complex natural logarithm) written in cartesian form - the same principle that is used in obtaining the standard real SLI representation.

The question now is how this should be represented as a computer word. We concentrate here on the equivalent of single precision SLI which uses a 32-bit word for real number representation. The complex format will therefore use 64-bits. This is the format which has been incorporated into the complex extension of the real Turbo Pascal SLIunit discussed in [16]. Of course, any wordlength is possible in principle and the relative lengths of the subwords used to represent the modulus and argument can also be varied. In the discussion below, we use two equal length subwords. Using a single word rather than two 32-bit words facilitates inline arithmetic in Turbo Pascal since the single 64-bit word can be the output of a function as opposed to a procedure.

For many calculations there is no significance in the order of magnitude of the phase but just its value modulo $2\pi$ and so there is no advantage in representing this angle in floating-point or any other variable precision format. For this reason therefore, we represent the argument in *fixed-point* form as a two's complement fraction of $\pi$. Thus the argument $\theta$ in (10) is represented by the integer $N_\theta$ satisfying $\theta = N_\theta\,\pi/2^{31}$ and $\theta \in [-\pi, \pi)$ .

This representation of the argument occupies the first 32 bits of the complex SLI, or CSLI, representation. The second part of the word contains the 32-bit SLI representation of the modulus. In the case of the Turbo Pascal software implementation, this is achieved by using the 64-bit integer type comp and the function defined by

```
function cpack(arg:longint; zmod:slisingle): slicomplex;
begin cpack:=c32*arg + zmod; end;
```

where the type comp constant c32 is $2^{32}$ and arg, zmod store respectively the values of $N_\theta$ above and the SLI representation of $R$. The details of the Turbo Pascal representation of $R$ can be found in [16].

## 3. CSLI arithmetic algorithms

In the case of floating-point complex arithmetic using the Cartesian representation of the complex variables the addition/ subtraction operations are straightforward and require just two of the corresponding real arithmetic operations. Multiplication necessitates four real multiplies and two additions while division is usually achieved (effectively) by reciprocation and multiplication making the total real operation count six multiplies, three additions and two divides. On even a minimally parallel processor, addition or subtraction could be achieved in the same time as their real counterparts while multiplication could be performed using two operations each on two parallel multiply-accumulate units. Division would demand three parallel multiply-accumulate processors each performing two operations followed by the parallel real divides.

Using the polar form obviously would simplify the multiplication and division operations to their real counterparts for the moduli together with addition or subtraction of the arguments. In the representation outlined above, the manipulation of the arguments requires just integer arithmetic with wraparound. On the other hand, addition and subtraction are apparently much more complicated since we must "solve the triangle".

As we shall see, in the case of CSLI arithmetic the additional complication of the addition and subtraction algorithms is very slight while, of course, the advantages for multiplication and division are retained. In what follows we describe the arithmetic algorithms for these CSLI operations and also consider which parts can, in principle, be performed in parallel.

For multiplication and division, the regular SLI operations can be performed on the moduli simultaneously with the two's complement addition or subtraction of the arguments. Even for an entirely serial implementation the only additional cost is one integer add/ subtract operation which, in that situation, is a small fraction of the SLI operation time. Thus CSLI multiplication and division can be performed, as one would expect, in essentially the same time as their real counterparts.

We concentrate for most of this section on the case of CSLI add/subtract operations and see that the additional cost relative to the corresponding real operation is small.

Consider then the addition of two CSLI variables. The task is to find $Z_3 = Z_1 + Z_2$ where each is given in its CSLI form; that is we require

$$\begin{aligned} R_3 e^{i\theta_3} &= \phi(r_3) e^{i\theta_3} = \phi(r_1) e^{i\theta_1} + \phi(r_2) e^{i\theta_2} \\ &= R_1 e^{i\theta_1} + R_2 e^{i\theta_2} \end{aligned} \tag{11}$$

We assume throughout that $R_1 \geq R_2$. Recall from [16] that magnitude comparsion is a straightforward task in the real SLI representation since the type `slisingle` is identified with type `longint` in an order preserving manner. The situation is illustrated in Figure 3.1.

The addition algorithm reduces therefore to the determination of $R_3 = \phi(r_3)$ and $\theta_3$ which involves the "solution of the triangle" in Figure 3.1.

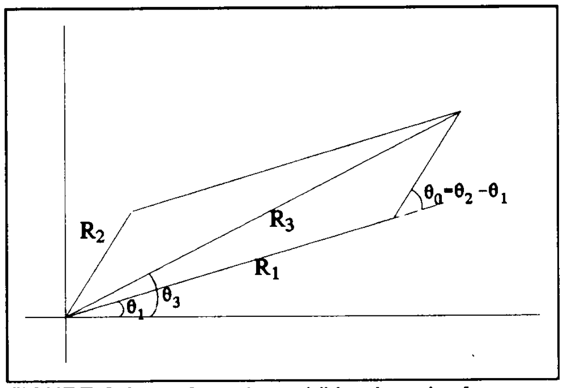

**FIGURE 3.1** Complex addition in polar form

Now, using the cosine rule we obtain

$$R_3^2 = R_1^2 + R_2^2 + 2R_1R_2\cos\theta_0$$

or, equivalently,

$$\frac{R_3^2}{R_1^2} = 1 + \frac{R_2^2}{R_1^2} + 2\frac{R_2}{R_1}\cos\theta_0$$

which, for the "large" case where $R_2 \geq 1$, reduces to

$$c_0^2 = 1 + b_0^2 + 2b_0\cos\theta_0 \tag{12}$$

There is no need to form the square-root of this quantity. Subsequent members of the $c$-sequence can be computed in the usual way with the minimal modification

$$c_1 = 1 + a_0 \ln c_0 = 1 + \frac{1}{2}a_0 \ln c_0^2$$

and, of course, the division by 2 can be achieved with a simple shift.

In just the same way, we obtain redefinitions of $c_0$ for the "mixed" ($R_1 \geq 1 > R_2$) and "small" ($R_1 < 1$) cases:

$$c_0^2 = 1 + (\alpha_0 a_0)^2 + 2(\alpha_0 a_0)\cos\theta_0$$
$$c_0'^2 = 1 + \beta_0^2 + 2\beta_0\cos\theta_0$$

where, in the small case, we use $c_0' = 1/c_0 = \phi(r_1)/\phi(r_3)$ just as for real SLI arithmetic.

The remainder of the $c$- and $h$- sequences can then be computed just as for real arithmetic as described in [4] or [16]. The questions which obviously arise here result from the need for trigonometric function evaluation within the algorithm. Before considering this in detail, we must examine the computation of the argument $\theta_3$ of the result.

The computation of the modulus is based on making a suitable adjustment to the larger operand - just as is the case with real SLI arithmetic. In the same way the argument will be computed as a correction to $\theta_1$. That is, we compute $\theta_3 - \theta_1$. From the trigonometry of the triangle in Figure 3.1, we see that

$$\begin{aligned} \theta_3 - \theta_1 &= \arctan\left[\frac{R_2\sin\theta_0}{R_1 + R_2\cos\theta_0}\right] \\ &= \arctan\left[\frac{b_0\sin\theta_0}{1 + b_0\cos\theta_0}\right] \end{aligned} \qquad (13)$$

for the large case. Similar expressions are appropriate for the other cases.

Subtraction can be handled by a simple adjustment of the argument of the subtrahend by the addition (or, equivalently, subtraction) of $\pi$. In the fixed-point two's complement fraction of $\pi$ representation being used, this is simply a reversal of the first (sign) bit of the stored argument. A simple sorting procedure is necessary to ensure that the operation is arranged so that $R_1 \geq R_2$. It may therefore be necessary to have a similar adjustment to the argument of the final result in the event of subtraction where the arguments have been reversed. This is again a single bit reversal.

We see that for all cases of the arithmetic, $\cos\theta_0$ is required for the computation of the modulus while this quantity and $\sin\theta_0$ are necessary for the argument computation. These two can be computed simultaneously with even minimal parallelism. Indeed the suggested hardware implementations discussed in [14], [17] make extensive use of modified CORDIC algorithms for computing the various sequences for SLI arithmetic and the CORDIC algorithms for trigonometric function evaluation naturally generate both values simultaneously. In either event the relative time penalty associated with this will be slight. Even within the software implementation, which, of necessity, uses the built-in general purpose exponential and logarithm functions, these trigonometric function calls represent a very small relative additional cost.

The only further cost involved in computing the modulus comes from the multiplication of $b_0$ by both itself and by $\cos\theta_0$ and the addition of these terms. With even a small degree of parallelism available, this represents the only true additional cost since the trigonometric evaluations could then be performed simultaneously with (one term of) the $a$-sequence. There is even some compensatory simplification in the algorithm resulting from the fact that both moduli are necessarily positive.

It is worthwhile noting here that the internal computation of the SLI and CSLI algorithms is entirely *fixed-point* in nature so that these additional costs would indeed be very small in a hardware implementation of CSLI arithmetic. Furthermore, the arguments are restricted to the interval $[-\pi, \pi)$ and are stored in two's complement integer form. It follows that the need for range reduction is almost eliminated. Detecting the appropriate quadrant, for example, is reduced to testing just the sign bit and the most significant bit. The interval of convergence of the CORDIC trigonometric routines is greater than $\pi/2$ and so determining the right quadrant is sufficient range reduction. Again the fact that fixed-point internal arithmetic is to be used renders the simplest, absolute precision, version of the CORDIC algorithm suitable.

It remains to complete the calculation of $\theta_3$. Once $b_0\cos\theta_0$, $b_0\sin\theta_0$ have been computed, all that is needed is the evaluation of the arctangent as in (13) and the (integer with wraparound) addition of $\theta_1$ to the result. The division implied by (13) need not be explicitly performed, we can instead use the CORDIC (or other) algorithm for arctan $(u/v)$ with $|u/v| \leq 1$. In the event that $|b_0\sin\theta_0| \geq |1 + b_0\cos\theta_0|$ then we are computing the arctangent of the reciprocal of the actual value; the resulting angle must be adjusted accordingly by subtraction from $\pi/2$.

The cost of this part of the algorithm is one additional trigonometric function evaluation and one or possibly two fixed point additions. For the serial software implementation, this is a very small additional cost - indeed the overall cost of complex addition in this situation is much *less* than the doubling which would be expected for standard cartesian representation. For any parallel implementation - and especially for the sort of hardware implementation envisioned in [16] - there would be no additional cost in computing the argument and so the overall increase would be just the multiplications needed for the formation of $c_0^2$.

Thus, in the SLI environment, the polar form of complex arithmetic - which is more appropriate for many computations - is, at worst, only very slightly slower than its real counterpart. This contrasts markedly with the (cartesian) floating-point situation where addition and subtraction have a cost factor of two, while for multiplication and division the factor grows to 6 or more. Of course, even in a hardware implementation, we expect the underlying SLI arithmetic to be somewhat slower than for floating-point - but this is yet another area in which that time loss is eroded for more advanced operations.

In the next section discuss the precision requirements for these algorithms, but first we consider some of the other binary and unary CSLI operations which are necessary for successful complex scientific computing.

Most of the unary operations require very simple manipulation after the 64-bit word is unpacked into its modulus and argument. We have already seen from considering subtraction that the unary minus operation involves simply a reversal of the first bit of the argument. Obtaining the modulus is achieved by the unpacking operation itself although there are two possible forms in which the result may be required - either as a real SLI number or as a CSLI quantity with argument zero. Either is simple to achieve and both are included in the software implementation.

Formation of the complex conjugate is achieved by a two's complement operation on the argument. Reciprocation involves this same two's complementing of the argument together with the real SLI reciprocation of the modulus which with the certainty of positivity is especially easy - consisting of just the reversal of all but the first (sign) bit of the SLI modulus.

Other binary operations are important parts of complex computation. The formation of integer roots of complex variables is readily achieved by simply dividing the argument by the appropriate integer and forming the required root of the SLI modulus - a straightforward operation. The representation of the argument is such that this necessarily delivers the principal value of the desired complex root. Others can be readily obtained by simple rotations of this one. Rotation through any angle can be achieved by the simple fixed-point addition of the angle and the argument.

For the case of FFT calculations rotations through angles of the form $k\pi/2^n$ are needed. Provided $n < 31$ - which is of course true for all practical cases - then this corresponds to the addition of the integer $k2^{31-n}$ to the argument of the complex operand. Thus such a rotation amounts to no more than a shift-and-add operation in the argument.

Forming integer powers of complex SLI variables is similarly straightforward necessitating only the corresponding SLI-integer operation together with the integer multiplication of the argument. Some economy is also available for other mixed operations.

In the case of addition or subtraction of a real SLI or integer variable to a complex SLI quantity, the real operand is simply converted to CSLI form and the arithmetic performed in the manner outlined above. (Some very minor simplification of this is possible in principle but does not yield sufficient saving to justify the special algorithms.) For multiplication and division of course the situation is somewhat different. Either the standard SLI multiplication/ division algorithm or the SLI-integer operation is used on the moduli while the argument is left unchanged for multiplication or division by a real positive quantity. For negative real operands, the argument must be adjusted by $\pi$ while division by the CSLI variable necessitates negation of the argument. Both of these operations have already been seen to be especially simple for the CSLI representation described here.

The representation proposed is also convenient for some of the elementary functions. Complex logarithms are especially simple:

$$\ln(\phi(r)^{\pm1}e^{i\theta}) = \pm\phi(r-1) + i\theta \tag{14}$$

This result must be converted to the polar CSLI form using algorithms similar to, but simpler than, those required for input conversion. This conversion automatically gives the principal value of the complex logarithm.

The complex exponential function is similarly straightforward to define for the CSLI representation:

$$\exp(\phi(r)^{\pm1}e^{i\theta}) = e^{\phi(r)^{\pm1}\cos\theta}\, e^{i\phi(r)^{\pm1}\sin\theta} \tag{15}$$

The first factor here is the modulus which can be computed by a modified version of the real SLI exponential function routine. The second factor yields the argument which must be reduced to its principal value.

## 4. Precision analysis

In this section we consider the internal accuracy required for the various parts of the CSLI arithmetic algorithm in order to control the errors to be of the order of inherent error. This is based on a similar linearized error analysis to that used for the regular LI and SLI algorithms in [2], [4] and [17]. As in those cases, we find that working to fixed *absolute* precision in the internal arithmetic will suffice. As in [17], denote by $\gamma_1$ and $\gamma_2$ the precision used for the *a*- and *b*- sequences respectively.

We shall only consider, in any detail, the case of "large" arithmetic in which $R_1, R_3 \geq 1$. We consider first the computation of the modulus. in order to obtain the required control over the error we must first consider the inherent error for this computation. From the cosine rule for the triangle of Figure 3.1, we have

$$\phi(r_3)^2 = \phi(r_1)^2 + \phi(r_2)^2 + 2\phi(r_1)\phi(r_2)\cos\theta_0 \tag{16}$$

from which we obtain the first-order estimate of the error $\delta r_3$ in $r_3$

$$\delta r_3 \approx \frac{1}{\phi(r_3)\phi'(r_3)}\left\{\begin{array}{l}[\phi(r_1)+\phi(r_2)\cos\theta_0]\phi'(r_1)\delta r_1 \\ +[\phi(r_2)+\phi(r_1)\cos\theta_0]\phi'(r_2)\delta r_2 \\ +\phi(r_1)\phi(r_2)\sin\theta_0\delta\theta_0\end{array}\right\} \tag{17}$$

This is, of course, a first-order estimate of the *generalized precision* of the modulus of the result. The dominant term in (17) is $[\phi(r_1)\phi'(r_1)/\phi(r_3)\phi'(r_3)]\,\delta r_1$ which is to be compared with the dominant term in the corresponding real SLI inherent error $[\phi'(r_1)/\phi'(r_3)]\,\delta r_1$.

For the "additive" case, $\phi(r_3) \le 2\phi(r_1)$ and so the complex error is at least one half of the real error. It follows that controlling the error to the same order of magnitude as in the real case will suffice for the complex algorithm. Similarly, in the "subtractive" case where $\phi(r_3) \le \phi(r_1)$ the complex error term is greater than its real counterpart and so achieving the same error control as for real SLI arithmetic will again suffice.

Now the only additional source of error in this calculation relative to the standard real SLI addition operation comes from the definition of $c_0^2$ in (12). We obtain

$$\begin{aligned}|\delta c_0^2| &\le |2\delta b_0\cos\theta_0 + 2b_0(\delta\cos\theta_0 + \delta b_0)| \\ &\le 4\delta b_0 + 2\delta\cos\theta_0\end{aligned} \tag{18}$$

since $|b_0|, |\cos\theta_0| \le 1$. The remainder of the arithmetic algorithm proceeds just as in the real case. The bound (18) compares with $|\delta c_0| = |\delta b_0|$ for the real operations. If the trigonometric functions are evaluated to the greater precision of $\gamma_1$ then the error at this stage of the CSLI algorithm is essentially 4 times that for the real SLI algorithm. It follows that increasing the precision $\gamma_2$ by two more bits allows the same accuracy to be achieved as for real SLI arithmetic.

We turn now to the computation of the argument correction $\theta_3$-$\theta_1$ and the control of the error there. Again we must first identify the dominant term inherent in the first-order error for this computation. There is one special case which should be isolated from the discussion, namely the situation of subtraction of two equal complex quantities. This case would be characterized in the algorithm by the conditions $b_0 = 1$ and $\cos\theta_0 = 1$.

We have

$$\begin{aligned}\hat\theta = \theta_3 - \theta_1 &= \arctan\frac{\phi(r_2)\sin\theta_0}{\phi(r_1 + \phi(r_2)\cos\theta_0} \\ &= \arctan\frac{b_0\sin\theta_0}{1+b_0\cos\theta_0} = \arctan\frac{u}{v}\end{aligned} \tag{19}$$

from which we may deduce

$$\delta\hat\theta \approx \frac{v\delta u - u\delta v}{u^2+v^2} \tag{20}$$

and, using absolute values throughout, we get

$$\begin{aligned}\delta u &\approx \sin\theta_0\,\delta b_0 + b_0\cos\theta_0\,\delta\theta_0 \\ &\approx \sin\theta_0\left[\frac{\phi'(r_2)}{\phi(r_1)}\delta r_2 + \frac{\phi(r_2)\phi'(r_1)}{(\phi(r_1))^2}\delta r_1\right] \\ &\qquad + \frac{\phi(r_2)}{\phi(r_1)}\cos\theta_0\,\delta\theta_0 \\ &= b_0\sin\theta_0\left[\phi'(r_2-1)\delta r_2 + \phi'(r_1-1)\delta r_1\right] \\ &\qquad + b_0\cos\theta_0\,\delta\theta_0\end{aligned} \tag{21}$$

and

$$\begin{aligned}\delta v &\approx \cos\theta_0\left[\frac{\phi'(r_2)}{\phi(r_1)}\delta r_2 + \frac{\phi(r_2)\phi'(r_1)}{(\phi(r_1))^2}\delta r_1\right] \\ &\qquad + \frac{\phi(r_2)}{\phi(r_1)}\sin\theta_0\,\delta\theta_0 \\ &= b_0\cos\theta_0\left[\phi'(r_2-1)\delta r_2 + \phi'(r_1-1)\delta r_1\right] \\ &\qquad + b_0\sin\theta_0\,\delta\theta_0\end{aligned} \tag{22}$$

There are again two cases to consider. These are easily defined by the conditions $|u| \le |v|$, in which case the argument adjustment satisfies $\hat\theta \le \pi/4$ and $|u| > |v|$, with $\hat\theta > \pi/4$. These do not correspond directly to the two cases for the modulus analysis above.

In the first case, the dominant term in (20) is approximately $(1/v)\delta u$ which, using the dominant term of (21), yields

$$\delta\hat\theta \approx \frac{b_0\sin\theta_0\phi'(r_1-1)}{1+b_0\cos\theta_0}\delta r_1 \le \phi'(r_1-1)\delta r_1$$

In the same manner, we obtain the inherent error estimate for $b_0$

$$\delta b_0 \approx \frac{\phi'(r_1)\phi(r_2)}{(\phi(r_1)^2)}\delta r_1 = \phi'(r_1-1)\delta r_1$$

It follows that the same working precisions as are used in [2] for regular LI arithmetic control the error in the argument correction to a similar level. The bound obtained there is

$$\delta b_0 \le 2.4\gamma_2\phi'(r_1-1)$$

and with the additional two bits accuracy suggested above for the modulus calculation giving $\gamma_2 = 2^{-33}$ this yields the desired accuracy in $\hat\theta$.

The second case in which $|u| > |v|$ includes the situation of extreme cancellation when $b_0 \approx 1$, and $\theta_0 \approx \pi$. We may assume without loss of generality that $\theta_0 > 0$. Now for this case, the dominant term in (19) is

$$\delta\theta \approx \frac{1}{u}\delta v \approx \cot\theta_0 \phi'(r_1 - 1)\delta r_1 \tag{23}$$

using (22) also. The error estimate (23) clearly grows as $\theta_0 \to \pi$ as would be expected. (It also grows as $\theta_0 \to 0$ but of course that would also imply $u < v$ and so is not included in the present case.)

Now the computed value has the first-order error estimate

$$\begin{aligned}\delta\hat{\theta} &\approx \frac{v\delta u - u\delta v}{u^2+v^2} + \gamma_3 \\ &\le (1/u)(|\delta u| + |\delta v|) + \gamma_3 \\ &\le \frac{1}{b_0\sin\theta_0}\left[\begin{array}{l}|\delta b_0 \sin\theta_0| + |b_0\cos\theta_0\delta\theta_0| \\ + |\delta b_0\cos\theta_0| + |b_0\sin\theta_0\delta\theta_0|\end{array}\right] + \gamma_3 \\ &= \frac{1+|\cot\theta_0|}{b_0}\delta b_0 + (1+|\cot\theta_0|)\delta\theta_0 + \gamma_3\end{aligned} \tag{24}$$

where $\gamma_3$ is the working precision of the arctangent computation.

Now for $u > v$, we have $1 + b_0\cos\theta_0 - b_0\sin\theta_0 < 0$ and this function has its minimum with respect to $\theta_0$ at $3\pi/4$ and its value there is $1 - b_0\sqrt{2}$ which can only be negative for $b_0 > 1/\sqrt{2}$. It follows that the estimate (24) yields

$$\begin{aligned}\delta\hat{\theta} &\le \sqrt{2}(1+|\cot\theta_0|)\delta b_0 \\ &\quad + (1+|\cot\theta_0|)\delta\theta_0 + \gamma_3 \\ &\le \sqrt{2}(1+|\cot\theta_0|)2.4\gamma_2\phi'(r_1-1) \\ &\quad + (1+|\cot\theta_0|)\delta\theta_0 + \gamma_3\end{aligned} \tag{25}$$

For all cases of interest, $|\cot\theta_0| \ge 1$ and since $\phi'(r_1-1) \ge 1$, the estimate (25) can in turn be bounded by

$$\delta\hat{\theta} \le |\cot\theta_0|\phi'(r_1-1)(6.8\gamma_2 + \gamma_3 + 2\delta\theta_0) \tag{26}$$

The representation of the argument as a two's complement fixed-point fraction of $\pi$ has a rounding unit of $\pi 2^{-31}$ which is therefore the appropriate quantity to use for $\delta\theta_0$ in (26). The working precision $\gamma_2 = 2^{-33}$ was used above and taking $\gamma_3 = 2^{-31}$ as a suitable precision for the arctangent routine, we obtain from (26)

$$\begin{aligned}\delta\hat{\theta} &\le |\cot\theta_0|\phi'(r_1-1)(6.8\times 2^{-33} + 2^{-31} + \pi\times 2^{-30}) \\ &\approx 4.0\times 2^{-30}|\cot\theta_0|\phi'(r_1-1) \\ &= 2^{-28}|\cot\theta_0|\phi'(r_1-1)\end{aligned}$$

Comparing this with (23) and remarking that the rounding unit for the single precision SLI representation is $2^{-28}$, we see that the desired error control has indeed been achieved.

Similar conclusions can be obtained in a similar manner for the "mixed" and "small" cases of CSLI addition and subtraction. The details are omitted here.

## 5. Conclusions

In this paper we have presented details of the symmetric level-index representation of complex numbers in modulus argument form. The virtues of this polar form for many scientific computing tasks make this representation philosophically attractive. The CSLI arithmetic algorithms presented show that this attractiveness carries over to the practical.

The multiplicative operations are of course simplified but we also see that addition and subtraction of CSLI quantities is only marginally more complicated than for the corresponding real operations. A hardware implementation of CSLI arithmetic would be expected to be only unnoticeably slower than real SLI arithmetic using the same technology.

It has also been established that arithmetic errors can be controlled to be of the same magnitude as the inherent error using only very slightly improved internal working precisions - but retaining the desirable use of just *fixed-point, fixed-precision* internal computation.

The particular representation and algorithms discussed have been implemented in software as an extension to the Turbo Pascal SLI unit. The representation appears to lend itself naturally to calculations such as Fast Fourier Transforms. This and other applications will be tested and the results reported on subsequently.

### References

[1] C.W.Clenshaw and F.W.J.Olver, *Beyond floating point*, J. ACM 31 (1984) 319-328.

[2] C.W.Clenshaw and F.W.J.Olver, *Level-index arithmetic operations*, SIAM J Num Anal 24 (1987) 470-485.

[3] C.W.Clenshaw, F.W.J.Olver and P.R.Turner, *Level-index arithmetic: An introductory survey*, Numerical Analysis and Parallel Processing (P.R.Turner Ed.) Lecture Notes in Mathematics 1397, Springer Verlag, 1989, pp. 95-168.

[4] C.W.Clenshaw and P.R.Turner, *The symmetric level-index system*, IMA J Num Anal 8 (1988) 517-526.

[5] C.W.Clenshaw and P.R.Turner, *Root-squaring using level-index arithmetic*, Computing 43 (1989) 171-185.

[6] H.Hamada *URR: Universal representation of real numbers*, New Generation Computing, 1 (1983) 205-209.

[7] H.Hamada, *A new real number representation and its operation*, pp. 153-157, Proc. ARITH7, (M.J.Irwin and R.Stefanelli, Eds) IEEE Computer Society, Washington DC, May 1987.

[8] D.W.Lozier and F.W.J.Olver, *Closure and precision in level-index arithmetic*, SIAM J Num. Anal, 27 (1990) 1295-1304.

[9] D.W.Lozier and P.R.Turner, *Robust parallel computation in floating-point and SLI arithmetic*, Computing 48 (1992) 239-257.

[10] S.Matsui and M.Iri *An overflow/underflow-free floating -point representation of numbers*, J. Information Proc. 4 (1981) 123-133

[11] F.W.J.Olver, *A new approach to error arithmetic*, SIAM J Num Anal. 15 (1978) 368-393

[12] F.W.J.Olver, *Rounding errors in algebraic processes - in level-index arithmetic*, Proc. Reliable Numerical Computation (M.G.Cox and S.Hammarling, Eds) Oxford, 1990, pp.197-205.

[13] F.W.J.Olver and P.R.Turner, *Implementation of level-index arithmetic using partial table look-up*, Proc. ARITH8, (M.J.Irwin and R.Stefanelli, Eds) IEEE Computer Society, Washington, DC, 1987, 144-147.

[14] P.R.Turner, *Towards a fast implementation of level-index arithmetic*, Bull IMA 22 (1986) 188-191.

[15] P.R.Turner, *Algorithms for the elementary functions in level-index arithmetic*, Scientific Software Systems (M.G.Cox and J.C.Mason Eds) Chapman and Hall, 1990, pp. 123-134.

[16] P.R.Turner, *A software implementation of sli arithmetic*, pp. 18-24, Proc. ARITH9, (M.D.Ercegovac and E.Swartzlander, Eds) IEEE Computer Society, Washington DC, September 1989.

[17] P.R.Turner, *Implementation and analysis of extended SLI operations*, pp. 118-126, Proc. ARITH10 (P.Kornerup and D.W.Matula, Eds) IEEE Computer Society, Washington DC, June 1991.

[18] H.Yokoo, *Overflow/underflow-free floating-point number representations with self-delimiting variable length exponent field*, pp. 110-117, Proc. ARITH10 (P.Kornerup and D.W.Matula, Eds) IEEE Computer Society, Washington DC, June 1991.