

A SOFTWARE IMPLEMENTATION OF SLI ARITHMETIC

Peter R Turner

Mathematics Department
US Naval Academy
Annapolis, MD 21402

ABSTRACT

In this paper we describe an implementation of the symmetric level-index, *sli*, system, some of its special features and some computational experience with it. The particular implementation discussed was developed for use on IBM-compatible PC machines and is written in Turbo PASCAL. This allows many of the attractive features of a potential hardware implementation of *sli* arithmetic to be readily incorporated. The ease of performing extended computational operations, such as scalar products and evaluation of polynomials, is evident from the package. The computational experiments reported here also show the great simplicity of program structure which this robust arithmetic permits.

1.

Introduction

This paper is largely concerned with implementational details of the symmetric level index, *sli*, system for computer arithmetic. This system which evolved from the original level-index, *li*, system introduced by Clenshaw and Olver [2], [3] is described in detail in [5]. The particular implementation discussed is contained in a Turbo PASCAL unit for implementation on IBM-compatible PC's which is available from the author. The choice of this programming language is justified by the great flexibility it offers for bit manipulation and creating new data types. Some of this justification will become apparent shortly. There are other software implementations of the *li* and *sli* systems, typically in FORTRAN, and results of numerical experiments with these have been reported elsewhere. (See [3] and [6] for example.) Possible hardware implementations have been discussed in [7] and [9]. The principal motivation for the present work is to obtain further insight into the eventual hardware design whilst providing a convenient vehicle for further computational experimentation.

The implementation under discussion incorporates much more than just the basic arithmetic operations. The very nature of the level-index representation (which we review briefly below) makes it entirely natural to incorporate the operations of forming powers - both integral and nonintegral - and roots. Also included are the standard elementary functions as well as *sli* vectors and their scalar products and norms.

In [10], it was stated that the evaluation of polynomial functions can be particularly efficient in the *sli* system. This claim is substantiated by the inclusion of this operation as a built-in function. This, and the other special features of the implementation are discussed in detail in Section 3.

The extension of the euclidean vector norm to the more general *p*-norms is considered in Section 4 on computational experience. The importance of this operation arises not just from mathematical interest but more significantly from the need to be able to compute these quantities efficiently for some of the recently developed and highly promising approaches to multivariate approximation using radial basis functions. (An

introduction to this topic can be found in [8].) This particular operation goes a long way towards answering some of the (unsubstantiated) criticisms of the level-index systems. The operations **could not be performed** in floating-point arithmetic. (A quick look at Blue's algorithm [1] for the euclidean norm will convince anyone that it will be an immense task to cope with the more general *p*-norm problem.) However the *sli* implementation of this operation is entirely straightforward **and accurate**. This latter claim is justified by demonstrating the convergence of the *p*-norm of a vector to the maximum or supremum norm as *p* increases.

We begin with a brief review of the *li* and *sli* representations and a general description of the computer package.

2.

Review and description of the implementation

A positive number *X* is represented in the *li* system by *x* where

$$X = \phi(x) \quad (2.1)$$

and the *generalized exponential function* ϕ is defined, for positive arguments by

$$\phi(x) = \begin{cases} x & 0 \leq x \leq 1, \\ \exp(\phi(x-1)) & x > 1. \end{cases} \quad (2.2)$$

It follows that $x = l + f$ where *l*, the *level*, is a nonnegative integer and the *index* $f \in [0, 1)$ is given by

$$f = \ln(\ln(\dots(\ln X) \dots)), \quad (2.3)$$

the natural logarithm being taken *l* times.

In the symmetric level-index system, a number in the interval (0, 1) is represented by the *li* image of its reciprocal. Thus a real number *X* can be represented by

$$X = \pm \phi(x)^{\pm 1}. \quad (2.4)$$

The arithmetic algorithms for the *li* and *sli* systems are described in detail in [3] and [5] and possible schemes for their hardware implementation were discussed in [7] at ARITH8.

Before proceeding to the arithmetic details it is worthwhile to discuss briefly the representation used within the software unit since this is one instance in which greater insight into the eventual hardware configuration is achieved. On conversion from floating-point input to symmetric level-index form the separate pieces of the representation are obtained and then packed into a single 32-bit long-integer format with the sign and reciprocation sign occupying the two most significant bits followed by the three bit level and the index. This is achieved in the procedure Pack whose Turbo PASCAL code is listed below.

```

Procedure Pack(psi: slirec; var x: slisingle);
{ Packs the sli record form into one
  32 bit integer variable }
{ Uses 1's complement forms to preserve
  usual integer ordering }

var lx : longint;
begin
  with psi do begin
    lx := level;
    x := (lx shl 27) or index;
    if r then x := x or c30 else x := not(x shl 2) shr 2;
    { INSERT RECIPROCATION SIGN AND
      1's COMPLEMENT LEVEL AND INDEX
      FOR RECIPROCAL FORM }
    if s then x := not(x);
    { 1's COMPLEMENT FORM FOR NEGATIVES }
  end;
end;

```

As usual the principal sign bit takes the value 1 for negative and 0 for positive quantities. The representation of $-X$ is simply the 1's complement of that of X . The reciprocation sign takes the value 1 for quantities with absolute value greater than unity. In the case of quantities with absolute value less than 1, the representation of the level and index is also negated. This has the effect of preserving the natural ordering of the integer representation used.

This procedure also illustrates some of the reasons for choosing Turbo PASCAL as the programming language. The use of the boolean operations and shifts renders the coding of this procedure very similar to the hardware instructions which would be needed for this representation. The two variable types in the procedure declaration, *slirec* and *slisingle* are defined in the main part of the unit. *Slirec* is a *record* consisting of two boolean variables for the sign and reciprocation sign, a byte variable to hold the level and finally a "longint" for the index. (Of course the index should be a fixed-point fraction and so this integer quantity must be shifted accordingly for the arithmetic.) The variable type *slisingle* is simply a long-integer which is being used for the storage of sli data in the manner outlined above. The quantity *c30* used in the procedure is one of several such longint variables which are used in the package; it is defined to be $1 \text{ shl } 30$ so that it is equivalent to

2^{30} and is used for insertion or detection of the reciprocation sign.

A corresponding procedure "Unpack" is used to obtain all the individual components of an sli number using similarly simple bit-pattern manipulations.

The algorithms for li and sli arithmetic are based on the computation of the terms of three short sequences. For example, the case of computing z such that

$$\phi(z) = \phi(x) + \phi(y) \quad (x \geq y) \quad (2.5)$$

uses the sequences

$$a_j = 1 / \phi(x-j), \quad b_j = \phi(y-j) / \phi(x-j), \quad (2.6)$$

$$\text{and} \quad c_j = \phi(z-j) / \phi(x-j)$$

which are computed from appropriate starting values by recurrence relations. In the implementation under present consideration, this computation uses the built-in logarithm and exponential functions. (This would not be the most efficient approach in a hardware implementation; see [7] and [9] for possible methods to be used in hardware.) In addition to the usual arithmetic operations, the level-index systems lend themselves to the natural inclusion of exponentiation as part of

the basic arithmetic package. Indeed the computation of $\phi(x)^{\phi(y)}$ is typically a slightly simpler and quicker operation than any of the four standard operations.

Also included in the Turbo PASCAL unit are all the mixed integer-sli arithmetic operations. One of the criticisms which has been levelled against the level-index system is that integers are not represented exactly within it. Of course, just as with floating-point arithmetic, it would be essential to have integer variable types alongside the sli variables. What is then important is not whether small integers can be *represented* exactly within the scheme but rather that they should be used exactly within the arithmetic. This is true of the sli system **but not of floating-point**.

Typically, within a floating-point system any integer variables which are used in arithmetic with floating-point operands as well are first converted into floating-point form. Either at this stage or in the subsequent alignment shifts the exactness of the representation is frequently lost. In sli arithmetic, however, the integer is used in its original form. For example, in the operation

$$\phi(z) = \phi(x) + n \quad (2.7)$$

we still compute the sequence $\{a_j\}$ but, instead of setting

$$c_0 = 1 + b_0$$

as would be the case for (2.5), we now put

$$c_0 = 1 + n a_0$$

and the rest of the computation proceeds just as normal having used the integer n exactly.

In a similarly simple manner, the other arithmetic operations involving a mixture of sli and integer operands including powers and roots are incorporated into the package. The elimination of the b -sequence for these mixed operations means that for a serial machine the arithmetic times for such operations will be approximately 30% less than for the corresponding full sli operations. (Again this contrasts with the floating-point situation in which the integer-to-floating-point conversion costs additional time.)

In [3] it was pointed out that the computation of extended sums can be performed efficiently within the li system and similar comments apply to the sli system. The gain in efficiency derives from the fact that the whole calculation can be based on the largest (in absolute value) term in the sum. This feature is exploited by the built-in function "VectorSum" in the Turbo PASCAL unit. The details are discussed in the next section but the principal point is simply that just one a -sequence (for that largest term) and one c -sequence are needed in order to form the sum of any number of terms. Furthermore, if sufficient parallelism is available, then all the remaining sequences can be computed simultaneously with the a -sequence reducing the operation time to just one standard arithmetic operation time. The corresponding floating-point operation requires of the order of $\log_2 n$ operation times, where

n is the number of terms being summed. It is clear that the slower basic operation times of the sli system could easily yield much faster overall operation times for extended sums.

As we shall see in the remaining sections of the paper, this same economy of computational effort can be achieved for computation of vector norms and the evaluation of monomials - and even polynomials. (A different, but similarly simple, routine is easily written for the computation of a "sum of squares" such as might frequently be required in the solution of systems of nonlinear equations: the routine included in the present package genuinely computes the euclidean norm - not its square.) Not only is the design of scientific software greatly simplified with sli arithmetic, it is very likely that the overall savings resulting from simplifications such as those alluded to

here will even reduce program run-times.

To give some indication of the ways in which these savings can be made we mention briefly here the case of the evaluation of a simple monomial term and then the computation of the sum of the squares of two symmetric level-index quantities.

For the former we require the value z for which

$$\phi(z) = \phi(y) * \phi(x)^n \quad (2.8)$$

or similarly defined quantities in the cases where any of the elements is in reciprocal form. For simplicity we consider the case of (2.8) with $x \geq y$. (All the other variations are similarly dealt with.) On taking logarithms, (2.8) becomes

$$\phi(z-1) = n * \phi(x-1) + \phi(y-1) \quad (2.9)$$

and division of this equation by $\phi(x-1)$ now yields the appropriate starting value for the c -sequence, namely

$$c_1 = n + b_1' \quad (2.10)$$

It is plain that this operation is no more complicated than any of the four standard arithmetic operations with these same sli

operands. For an algebraic monomial of the form $m \phi(x)^n$ the corresponding value of c_1 is $n + m a_1$.

The case of the sum of two squares again reduces to a simple adjustment of the starting value of the c -sequence as follows. Again we describe just one case whose simplicity is entirely typical of the operation, namely the calculation of z such that

$$\phi(z) = \phi(x)^2 + \phi(y)^2 \quad (x \geq y). \quad (2.11)$$

Now, dividing by $\phi(x)^2$ and taking logarithms, we get

$$\begin{aligned} \ln(\phi(z) / \phi(x)^2) &= \phi(z-1) - 2\phi(x-1) \\ &= \phi(x-1) (\phi(z-1) / \phi(x-1) - 2) \end{aligned}$$

from which it follows that

$$\begin{aligned} c_1 &= 2 + \ln(\phi(z) / \phi(x)^2) / \phi(x-1) \\ &= 2 + a_1 \ln c_0 \end{aligned} \quad (2.12)$$

where, for this case, c_0 is defined by

$$c_0 = \phi(z) / \phi(x)^2 = 1 + b_0^2. \quad (2.13)$$

The rest of the computation proceeds exactly as usual and the total extra cost is the apparent multiplication of the **fixed-point, fixed precision** fraction b_0 by itself. However even this

can be avoided by shifting the argument of the exponential function for the final step of the b -sequence one place to the left. The Turbo PASCAL unit does not include this last economy in its euclidean norm algorithm.

Again it is clear that as the arithmetic operations under consideration become, or appear to become, more complicated, so the simplifications offered by the new arithmetic grow. These simplifications also entail a very significant reduction in the number of round-off errors committed and are therefore likely to deliver greater accuracy in the final result of the computation. A detailed error analysis of extended sli arithmetic operations will appear elsewhere.

3. Important features

The principal features of the sli arithmetic package fall into three categories: scalar, vector and polynomial operations. In several of these it is immediately apparent just how great the benefit of suitable parallel processors would be. Some of these

benefits will be highlighted by considering operation counts.

Much of what is included in the scalar environment has been described in the various papers and is summarized in [4] along with results of some computational experience. However there are just a few points of implementational interest to be made.

The direct inclusion of mixed integer-sli arithmetic operations extends the range of elementary functions in one important way. In addition to the forming of integer powers of sli numbers, we have incorporated the operation of taking integer roots of any order. Again, this operation reduces to a simple redefinition of the initial value for the c -sequence. (Of course for PASCAL even the formation of integer powers is an extension of the normal library of built-in functions.)

All the sli arithmetic operations are incorporated into the function SLI which has the syntax $Sli(x, 'op', y)$ where x and y are variables of type slingle and op is one of $+$, $-$, $*$, $/$ and $^$. The integer-sli operations are incorporated into two functions depending on the order of the arguments; these have similar syntax and the same list of operations. The taking of roots is performed in a separate function, $Root(x, n)$, which forms the

n^{th} root of the slingle variable x . The square root is a special case, $SqRoot$, which simply calls $Root$ with $n = 2$.

At this stage the trigonometric functions are computed by using the corresponding floating-point functions. The risk of overflow errors here is completely acceptable since, as was observed in [10], the attempt to attribute a specific value to, say, $\cos \phi(x)$ for values x exceeding about 4 is meaningless as it will have no accuracy at all because the absolute error bound for $\phi(x)$ is at least $\pi/2$. Of the other standard elementary functions, the logarithmic and exponential functions are, of course, straightforward while the arctangent is reduced by the usual identities to the computation of $\arctan(a_0)$.

Many of the more important features of the package are to be found within the vector operations. Firstly, a variable type *slivector* is declared which consists of an array whose elements are of type slingle. The size of the array was originally limited to a subscript range of 0 through 100. This arbitrary upper limit can be very easily varied to allow any dimension which can be represented in one of Turbo PASCAL's integer types, the only penalty being the additional memory allocation which any larger range would incur. For our present purposes the limit of 100 is no restriction.

The type *slivector* is of course likely to be used for the terms of a series. For both this purpose and for the formation of scalar products, the function $SumVector(v, n)$ which sums the elements of a *slivector*, v , with slingle elements $v[0]$ through $v[n]$ and returns the result in slingle form. Both for this operation, and for more general purposes, it is necessary to identify the *llargestl* (that is, largest in absolute value) term in the sum on which to base the efficient computation. The fact that the representation preserves the natural ordering of the integers used makes this particular operation particularly simple and we do not discuss the details.

The great efficiency of this operation is due to the fact that if we seek the sum

$$\phi(x_0) \pm \phi(x_1) \pm \dots \pm \phi(x_n), \quad (3.1)$$

where, without loss of generality, we may assume

$$x_0 \geq x_k \quad (k = 1, 2, \dots, n)$$

then the appropriate value of c_0 is just

$$c_0 = 1 \pm b_{0,1} \pm b_{0,2} \pm \dots \pm b_{0,n} \quad (3.2)$$

where $b_{0,k} = \phi(x_k) / \phi(x_0)$.

Since all the internal computation is fixed absolute precision fixed-point arithmetic, the ordering of the terms in (3.2) is immaterial and no roundoff errors are committed. It follows that the summation is performed with just a single sli roundoff. Furthermore, if there is sufficient parallelism available, then all of these terms can be computed simultaneously with the result that the whole summation will take no longer than a single sli arithmetic operation save for the fixed-point additions in (3.2), which itself can be made very efficient by use of a tree of Carry Save Adders. Even with an entirely serial processor, the operation count would still be bounded above by $(1 + n/3)$ since the b-sequence is, typically, the shortest of the three. If the sum (3.1) includes terms which are of reciprocal form then, for these terms, $\pm 1/a_{0,k}$ is added

into c_0 , where $a_{0,k} = 1/\phi(x_k)$. Again this can be computed

simultaneously with the various b-sequences. The remaining case, in which all terms are in reciprocal form, can be similarly shortened.

For the efficient and most accurate computation of such a floating-point sum, a much more sophisticated ordering of the terms would be required and, for a serial processor, n full floating-point additions are necessary. It is clear that much of the time-loss of individual sli arithmetic operations is recouped in the case of extended sums.

Once we have seen the ease of computation of the SumVector function, the following code for the scalar product of two slivectors not only looks simple but is immediately recognizable as highly efficient. It is self-explanatory.

```
Function ScalarProd(u, v: slivector; dim: byte): slisingle;
var
  w: slivector;
  i: byte;
begin
  for i := 0 to dim do w[i] := sli(u[i], '*' , v[i]);
  ScalarProd := SumVector(w, dim);
end;
```

At this point it would be easy to define the L_2 norm

function for a slivector by just $\text{SqRoot}(\text{ScalarProd}(u, u, \text{dim}))$. Consider the operation counts for this definition. For a sufficiently parallel computer, there is the computation of the working slivector, w , which is one operation. This is followed by the SumVector and then the SqRoot. A total of three operation times is sufficient for the whole computation. On a serial machine, the corresponding figures, with $\text{dim} = n$, are $n+1$, for squaring the components, $1 + n/3$ for the summation and one more for the final square-root or $3 + 4n/3$ arithmetic operations in all. This **completely robust** computation requires no scaling and separating of components as a safeguard against overflow or underflow and must be compared with the very involved routines such as Blue's [1] which are necessary in the floating-point environment.

This may, at first sight, seem a highly attractive procedure to follow but there are great savings in efficiency that can be made which in fact reduce the overall algorithm to a slightly simplified form of the original SumVector function. The point here is that yet again a redefinition of the starting point of the c-sequence does everything. This derives from a similar approach to that discussed for the sum of two squares. For the simplest case, we set

$$c_0 = \sqrt{1 + b_{0,1}^2 + b_{0,2}^2 + \dots + b_{0,n}^2} \quad (3.3)$$

with corresponding modifications for the other cases. Since c_0 is used as an argument of the natural logarithm function, the

square-root need not be taken at this point but the result of that logarithm is shifted one place to the right. With the

corresponding simplifications in the calculation of the $b_{0,k}^2$ it is

apparent that this operation can be achieved in a single sli operation time for a parallel processor or, as with SumVector, in $1 + n/3$ such operation times for a serial machine.

Since, in the unlikely event that it would work, the simplest of floating-point routines for this calculation requires at the very least $2n+2$ floating-point operations, it follows that with the (almost certainly) achievable ratio of 6 : 1 between basic sli and floating-point operation times, then the sli routine will compute the euclidean norm of a vector more quickly than could floating-point. The non-parallelizability of robust algorithms for the floating-point computation makes the sli system a sure winner in any parallel environment.

The other area in which the sli computing environment differs significantly from the floating-point one is in the ease of evaluation of polynomials, and the consequent inclusion of procedures for this purpose. We have already described briefly the evaluation of monomial terms. One of the important considerations there is the relative magnitudes of the coefficient and the argument, with the computation being based on the larger one. Again this reduces to the appropriate definition of c_0 . The operation of evaluation of a polynomial

function is then achieved by the following very simple piece of code which again is self-explanatory.

```
Function Poly(coeff: slivector; degree: byte; x: slisingle):
slisingle;
var
  i: byte;
  term: slivector;
begin
  for i := 0 to degree do
    term[i] := monomial(coeff[i], x, i);
  Poly := SumVector(term, degree);
end;
```

On this occasion there is no easy way of abbreviating the computation further and the implied $2 + 4n/3$ operations for a polynomial of degree n on a serial machine or 2 for a parallel one are indeed correct. This still represents a considerable saving in terms of the number of operations even by comparison with Horner's rule. Of course the operations will themselves be slower but the savings indicated, combined with the available simplicity of program structure, may well make the eventual hardware implementation of symmetric level-index arithmetic faster than floating-point for this operation.

4. Computational experience

The principal application we consider here is the computation of the p-norm of a vector. As has been observed already, this is of both practical and theoretical interest but more importantly, for our present purposes, it provides an excellent example of a piece of computation which is straightforward in the level-index systems while being very difficult to perform efficiently in floating-point arithmetic. The results demonstrate not only the possible efficiency of computation with sli arithmetic but also the accuracy delivered by sli routines which have used some very large, or very small, numbers en route to finding a moderately sized final result.

It is on this latter point that some of the critics of the level-index system have focused in the mistaken belief that any

calculation which uses quantities which have little if any *relative* precision cannot be meaningful. The fallacy of this argument is demonstrated by the computation of the p-norms of vectors for increasing values of p since in this case the limit as $p \rightarrow \infty$ is easily computed directly. It is just the ∞ -norm, or supremum norm, which is obtained directly from the MaxComp procedure for finding the $\|largest\|$ component of a vector. (This is the same procedure as was used within the SumVector function.)

Three different p-norm function routines were written - two with integer values of p and one for p a slingle variable. We concentrate here on the integer case, save to report that the computation of the p-norm for nonintegral values of p was also a completely reliable calculation irrespective of whether p was small, moderate or very large.

Why two functions for integer p? The first is the obvious simple routine, called IntNormP which is computed by the following code

```
Function IntNormP(u: slivector; dim: byte; p: integer)
: slingle;
var
  i: byte;
begin
  for i := 0 to dim do u[i] := SliInt(SliAbs(u[i]), '^', p);
  IntNormP := Root(SumVector(u, dim), p);
end;
```

Again it is clear that for a parallel machine the first stage could all be done simultaneously reducing the overall operation to just three "sliops". It is immediately apparent that for large values of p the results of the powers in the for loop will easily reach levels of overflow or underflow for vectors whose components are of moderate size. In the case where all

components are in reciprocal form, they would all underflow to zero to yield a meaningless zero result for the norm for large values of p. As can be seen from the results in the tables below, the sli system had no difficulty in computing accurate results.

The second function routine, IntPNorm, uses the same approach as was described in Section 3 for the euclidean norm and can potentially reduce the parallel processor version to just one operation. In its software implementation it is only marginally more efficient than the first of these. What is important to observe in all of these results is that both functions produce sequences of values for the p-norm which are converging steadily to the correct limit. Furthermore, there is no loss of accuracy as the value of p grows. All of the runs were continued out to at least $p = 1000$ with no loss of precision.

By this point, sums of quantities of the order of $\phi(4.8)$ had been performed and then their 1000th root taken. Such numbers have decimal exponents of the order of 5000 - a long way beyond the limits of any floating-point system. This is achieved with the single length sli format and so only a 32-bit word.

The tests were performed on randomly generated vectors of dimension 21. Firstly with moderately large components, uniformly distributed in the interval $[-15\,000, 15\,000]$, then with a mix of small and moderate components in $[-2.5, 2.5]$ and thirdly with all small components generated in the interval $[0, 1]$. The results for the two functions are tabulated in Tables 4.1(a) - (c) respectively. The values $\|v\|_p$ are obtained by

converting the values of IntNormP from sli to floating-point form.

Tables 4.1

Dimension = 21				
(a)	Components in $[-15\,000, 15\,000]$			
p	$\ v\ _p$	IntNormP	IntPNorm	
1	1.48277 E+05	[3.907094] 1	[3.907094] 1	
2	3.75909 E+04	[3.856395] 1	[3.856395] 1	
3	2.46956 E+04	[3.838960] 1	[3.838960] 1	
4	2.03779 E+04	[3.830635] 1	[3.830635] 1	
5	1.83394 E+04	[3.825970] 1	[3.825970] 1	
6	1.71941 E+04	[3.823081] 1	[3.823081] 1	
7	1.64779 E+04	[3.821159] 1	[3.821159] 1	
8	1.59958 E+04	[3.819811] 1	[3.819811] 1	
9	1.56535 E+04	[3.818825] 1	[3.818825] 1	
10	1.54004 E+04	[3.818080] 1	[3.818080] 1	
20	1.45017 E+04	[3.815316] 1	[3.815316] 1	
30	1.42988 E+04	[3.814664] 1	[3.814664] 1	
40	1.42160 E+04	[3.814395] 1	[3.814395] 1	
50	1.41721 E+04	[3.814252] 1	[3.814252] 1	
100	1.41024 E+04	[3.814024] 1	[3.814023] 1	
150	1.40898 E+04	[3.813982] 1	[3.813982] 1	
200	1.40868 E+04	[3.813972] 1	[3.813972] 1	
300	1.40858 E+04	[3.813969] 1	[3.813969] 1	
400	1.40857 E+04	[3.813968] 1	[3.813968] 1	
500	1.40857 E+04	[3.813968] 1	[3.813968] 1	
∞	1.40857 E+04	[3.813968] 1		

(b) Components in $[-2.5, 2.5]$

p	$\ v\ _p$	IntNormP	IntPNorm
1	2.50065 E+01	[3.156245] 1	[3.156245] 1
2	6.27878 E+00	[2.608229] 1	[2.608229] 1
3	4.17182 E+00	[2.356521] 1	[2.356521] 1
4	3.48042 E+00	[2.220863] 1	[2.220863] 1
5	3.15608 E+00	[2.139181] 1	[2.139181] 1
6	2.97294 E+00	[2.085767] 1	[2.085767] 1
7	2.85693 E+00	[2.048551] 1	[2.048551] 1
8	2.77750 E+00	[2.021321] 1	[2.021321] 1
9	2.71998 E+00	[2.000626] 1	[2.000626] 1
10	2.67659 E+00	[1.984543] 1	[1.984543] 1
20	2.51267 E+00	[1.921347] 1	[1.921347] 1
30	2.47615 E+00	[1.906704] 1	[1.906704] 1
40	2.46510 E+00	[1.902233] 1	[1.902233] 1
50	2.46149 E+00	[1.900765] 1	[1.900765] 1
100	2.45959 E+00	[1.899996] 1	[1.899996] 1
150	2.45958 E+00	[1.899991] 1	[1.899991] 1
200	2.45958 E+00	[1.899991] 1	[1.899991] 1
250	2.45958 E+00	[1.899991] 1	[1.899991] 1
300	2.45958 E+00	[1.899991] 1	[1.899991] 1
∞	2.45958 E+00	[1.899991] 1	

(c) Small components in $[0, 1]$

p	$\ v\ _p$	IntNormP	IntPNorm
1	1.12437 E+01	[2.883689] 1	[2.883689] 1
2	2.80409 E+00	[2.030607] 1	[2.030607] 1
3	1.82533 E+00	[1.601762] 1	[1.601762] 1
4	1.49352 E+00	[1.401134] 1	[1.401134] 1
5	1.33366 E+00	[1.287929] 1	[1.287929] 1
6	1.24181 E+00	[1.216571] 1	[1.216571] 1
7	1.18314 E+00	[1.168171] 1	[1.168171] 1
8	1.14292 E+00	[1.133590] 1	[1.133590] 1
9	1.11395 E+00	[1.107908] 1	[1.107908] 1
10	1.09227 E+00	[1.088253] 1	[1.088253] 1
20	1.01336 E+00	[1.013267] 1	[1.013267] 1
30	9.95005 E-01	[1.005007] -1	[1.005007] -1
40	9.87711 E-01	[1.012365] -1	[1.012365] -1
50	9.84164 E-01	[1.015962] -1	[1.015962] -1
100	9.80091 E-01	[1.020110] -1	[1.020110] -1
150	9.79811 E-01	[1.020396] -1	[1.020396] -1
200	9.79786 E-01	[1.020421] -1	[1.020421] -1
250	9.79784 E-01	[1.020423] -1	[1.020423] -1
300	9.79784 E-01	[1.020423] -1	[1.020423] -1
350	9.79784 E-01	[1.020423] -1	[1.020423] -1
400	9.79784 E-01	[1.020423] -1	[1.020423] -1
∞	9.79784 E-01	[1.020423] -1	

5.

Conclusions

In this paper we have seen that the symmetric level-index system of number representation and arithmetic can be implemented in a tolerably efficient software package which allows the inclusion of several special features. The Turbo PASCAL language allows for easy storage of the representation within a single 32-bit computer word which preserves the natural ordering of the representing integers.

Many of the features which are desirable but difficult to engineer within a floating-point system - such as the formation of scalar products and computation of euclidean norms - are reduced to just a very simple sli operation. Sometimes even to just a single such arithmetic operation.

The ideas used there have been extended to the computation of the p-norm of a vector which provides convincing evidence of the ability of the new arithmetic to deliver highly accurate results at the end of calculations which have used numbers well outside the range of even double precision (or even Turbo PASCAL's *extended* 80-bit format).

The functions and procedures discussed here are available as a Turbo PASCAL (Version 5.0) unit which may be obtained from the author for further experimentation.

Acknowledgements

The author is pleased to acknowledge helpful discussions with C.W.Clenshaw, D.W.Lozier and F.W.J.Olver in connection with this work and with W.A.Light who was responsible for bringing the practical importance of the p-norm to my attention. J.L.Buchanan was instrumental in teaching me some of the features of Turbo PASCAL.

(Turbo PASCAL is a trade mark of Borland International Inc.)

REFERENCES

- [1] J.L.Blue, *A portable FORTRAN program to find the euclidean norm of a vector*, ACM Trans Math Software 4 (1978) 15-23.
- [2] C.W.Clenshaw and F.W.J.Olver, *Beyond floating point*, J. ACM 31 (1984) 319-328.
- [3] C.W.Clenshaw and F.W.J.Olver, *Level-index arithmetic operations*, SIAM J Num Anal 24 (1987) 470-485.
- [4] C.W.Clenshaw, F.W.J.Olver and P.R.Turner, *Level-index arithmetic: An introductory survey*, Proc. Numerical Analysis Summer School, Lancaster, 1987, Springer Verlag Lecture Notes in Mathematics, to appear, 1989.
- [5] C.W.Clenshaw and P.R.Turner, *The symmetric level-index system*, IMA J Num Anal 8 (1988) 517-526.
- [6] C.W.Clenshaw and P.R.Turner, *Root squaring using level-index arithmetic*, to appear.
- [7] 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, 1987, 144-147.
- [8] M.J.D.Powell, *Radial basis functions for multivariable interpolation: A review*, Algorithms for Approximation 143 - 167 (M.G.Cox and J.C.Mason, eds.) Oxford, 1987.
- [9] P.R.Turner, *Towards a fast implementation of level-index arithmetic*, Bull. IMA 22 (1986) 188-191.
- [10] P.R.Turner, *Algorithms for the elementary functions in level-index arithmetic*, Proc. Symposium on Scientific Software and Systems, RMCS Shrivenham, 1988 (M.G.Cox and J.C.Mason, eds.), to appear.