Nikhef-10-002
arXiv:1005.1481

A Portable Collection of Fortran Utility Routines

MBUTIL version 7.0

M. Botje*
Nikhef, Science Park 105, 1098XG Amsterdam, the Netherlands

December 12, 2018

Abstract

The MBUTIL library is a well documented collection of FORTRAN routines. Some of these are taken from the public libraries CERNLIB and NETLIB while others are privately developed for use in the QCD evolution program QCDNUM. The aim of this collection is to make QCDNUM independent of the availability of external libraries, and also to give easy access to a large set of general-purpose QCDNUM routines. Of particular interest are routines that partition a linear store into multidimensional arrays—these are the basis of the simple but effective and fast QCDNUM dynamic memory management—and routines for fast interpolation. Also included is a string formatter that can be used to process free-format datacards.

*email m.botje@nikhef.nl

Contents

1 Introduction

The MBUTIL package is an integral part of the QCDNUM distribution[1] and contains a pool of FORTRAN utility routines. Some of these are developed privately, some are taken from CERNLIB and some are picked-up from public source code repositories such as NETLIB[2].

The routines in MBUTIL are grouped into general purpose utilities (Section 2), floating-point comparisons (Section 3), vector operations (Section 4), fast solution of triangular and band systems (Section 5), pointer arithmetic in a linear store (Section 6), fast interpolation (Section 7), bitwise operations (Section 8), character string manipulations (Section 9) and a string formatter (Section 10).

It is worthwhile to have a look at Section 6 where routines are described that dynamically partition a linear store into multi-dimensional arrays, and allow you to achieve fast indexing in these arrays. Indeed, QCDNUM owes much of its flexibility and speed to these routines. Of interest may also be Section 7 (fast interpolation) and Section 10 where it is described how to use a string formatter to process free-format datacards.

The syntax of the MBUTIL calls is as follows

[1] `http://www.nikhef.nl/user/h24/qcdnum`
[2] `http://www.netlib.org`

```
xMB_name ( arguments )
```

where x = S for subroutines and x = L, I, R or D for logical, integer, real and double-precision functions, respectively. Output variables will be marked by an asterisk (*) and in-out variables by an exclamation mark (!). A function type must be declared in the calling routine unless this is taken care of by an implicit type declaration, thus:

```
logical  lval, lmb_function
lval = LMB_FUNCTION ( arguments )
```

Unless otherwise stated all floating point calculations are done in double precision. This implies that actual floating point arguments should be given in double precision format:

```
dval = dmb_gamma ( 3.D0 )      ! ok
dval = dmb_gamma ( 3.0  )      ! wrong!
```

2 Utility Routines

In this section we describe the MBUTIL utility programs given in Table 1.

Table 1: Utility routines in MBUTIL. Output variables are marked by an asterisk (*) and in-out variables by an exclamation mark (!). CERNLIB references are given in the first column.

CERNLIB	Subroutine or function	Description
C302	`DMB_GAMMA ( x )`	Gamma function
C332	`DMB_DILOG ( x )`	Dilogarithm
D401	`SMB_DERIV ( f, x, !del, *dfdx, *derr )`	Differentiation
D103	`DMB_GAUSS ( f, a, b, e )`	Gauss integration
F010	`SMB_DMINV ( n, !a, m, ir, *ierr )`	Matrix inversion
F010	`SMB_DMEQN ( n, a, m, ir, *ierr, k, b )`	Linear equations
F012	`SMB_DSINV ( n, !a, m, *ierr )`	Invert symmetric matrix
M103	`SMB_RSORT ( !rarr, n )`	Sort real array
	`SMB_ASORT ( !rarr, n, *m )`	Sort and weed real array
	`RMB_URAND ( !iy )`	Uniform random numbers

```
dval = DMB_GAMMA ( x )
```

Calculate the gamma function

$$\Gamma(x) = \int_0^\infty e^{-t}\, t^{\,x-1}\, dt \quad (x > 0).$$

The function `dmb_gamma` as well as `x` and `dval` should be declared `double precision` in the calling routine. Code taken from CERNLIB C302 (`dgamma`).

```
dval = DMB_DILOG ( x )
```

Calculate the dilogarithm

$$\mathrm{Li}_2(x) = -\int_0^x \frac{\ln|1-t|}{t}\,dt.$$

The function `dmb_dilog` as well as `x` and `dval` should be declared `double precision` in the calling routine. Code taken from CERNLIB C332 (`ddilog`).

```
call SMB_DERIV ( fun, x, !del, *dfdx, *erel )
```

Calculate the first derivative $f'(x)$. The derivative of f should exist at and in the neighbourhood of x. This is the responsibility of the user: output will be misleading if the function f is not well behaved. Code taken from CERNLIB D401 (`dderiv`).

- `fun` User supplied double precision function of one argument (x). Should be declared `external` in the calling routine.
- `x` Value of x where the derivative is calculated.
- `del` Scaling factor. Can be set to 1 on input and contains the last value of this factor on output (see the CERNLIB write-up).
- `dfdx` Estimate of f' on exit. Set to zero if the routine fails.
- `erel` Estimate of the relative error on f'. Set to one if the routine fails.

```
dval = DMB_GAUSS ( fun, a, b, epsi )
```

Calculate by Gauss quadrature the integral

$$I = \int_a^b f(x)\,dx.$$

In the calling routine the function `dmb_gauss`, all its arguments and `dval` should be declared `double precision`. Code taken from CERNLIB D103 (`dgauss`).

- `fun` User supplied double precision function of one argument (x). Should be declared `external` in the calling routine.
- `a`,`b` Integration limits.
- `epsi` Required accuracy of the numerical integration.

```
call SMB_DMINV ( n, arr, idim, ir, *ierr )
```

Calculate the inverse of an $n \times n$ matrix A. Code taken from CERNLIB F010 (`dinv`).

- `n` Dimension of the square matrix to be inverted.
- `arr` Array, declared in the calling routine as `double precision arr(idim,jdim)` with both `idim` $\geq$ `n` and `jdim` $\geq$ `n`. On entry the first $n \times n$ elements of `arr` should contain the matrix A. On exit these elements will correspond to A^{-1}, provided that A is not found to be singular (as signalled by the flag `ierr`).
- `idim` First dimension of `arr`.
- `ir` Integer array of at least `n` elements (working space).
- `ierr` Set to -1 if A is found to be singular, to 0 otherwise.

```
call SMB_DMEQN ( n, arr, idim, ir, *ierr, k, b )
```

Solve the matrix equation $Ax = b$ with multiple right-hand sides. Code taken from CERNLIB F010 (`deqn`).

`n` Dimension of the square matrix A.

`arr` Array, declared in the calling routine as `double precision arr(idim,jdim)` with both `idim` $\geq$ `n` and `jdim` $\geq$ `n`. On entry the first $n \times n$ elements of `arr` should contain the matrix A. On exit, A is destroyed.

`idim` First dimension of `arr`.

`ir` Integer array of at least `n` elements (working space).

`ierr` Set to -1 if A is found to be singular, to 0 otherwise.

`k` Second dimension of array `b`.

`b` Array, dimensioned `b(idim,k)` that contains on entry a set of k right-hand side vectors of dimension n, and on exit the set of k solution vectors x.

```
call SMB_DSINV ( n, arr, idim, *ierr )
```

Calculate the inverse of an $n \times n$ symmetric positive definite matrix A (i.e. a covariance matrix). Code taken from CERNLIB F012 (`dsinv`).

`n` Dimension of the square matrix to be inverted.

`arr` Array, declared in the calling routine as `double precision arr(idim,jdim)` with both `idim` $\geq$ `n` and `jdim` $\geq$ `n`. On entry the first $n \times n$ elements of `arr` should contain the matrix A. On exit these elements will correspond to A^{-1}, provided that A is found to be positive definite (as signalled by the flag `ierr`).

`idim` First dimension of `arr`.

`ierr` Set to 0 if A is found to be positive definite, to -1 otherwise.

```
call SMB_RSORT ( !rarr, n )
```

Sort the first n elements of the real array `rarr` in ascending order onto itself. On exit we thus have

$$A_1 \leq A_2 \leq \cdots \leq A_{n-1} \leq A_n.$$

Note that `rarr` should be declared `real` and not `double precision` in the calling routine. Code taken from CERNLIB M103 (`flpsor`).

```
call SMB_ASORT ( !rarr, n, *m )
```

Sort the first n elements of the real array `rarr` in ascending order onto itself but discard equal terms. On exit `rarr` contains a list of $m \leq n$ terms such that

$$A_1 < A_2 < \cdots < A_{m-1} < A_m.$$

The remaining $n - m$ elements are undefined. Notice that `rarr` should be declared `real` and not `double precision` in the calling routine.

```
rval = RMB_URAND ( !iy )
```

Return a (real) uniform random number in the interval $(0, 1)$. The integer `iy` should be initialised to an arbitrary value before the first call to `rmb_urand` but should not be altered by the calling routine in-between subsequent calls. Note that `rmb_urand` must be declared `real` in the calling routine. Code taken from NETLIB.

3 Floating-point Comparisons

A floating-point number that should be zero often misses the test for zero because it suffers from rounding errors that are larger than the underflow gap.[3] A solution to this problem is to artificially enlarge the size of this gap by the replacement

$$\text{test}(\ a = 0\ ) \quad \rightarrow \quad \text{test}(\ |a| < \epsilon\ )$$

with ϵ a few orders larger than the expected numerical error (typically 10^{-13} in QCDNUM).

The logical functions below provide floating-point comparisons within a tolerance `epsi`.

Logical function	Comparison	Floating-point comparison
`LMB_EQ ( a, b, epsi )`	$a = b$	$\|a-b\| \le \epsilon$
`LMB_NE ( a, b, epsi )`	$a \ne b$	$\|a-b\| > \epsilon$
`LMB_GE ( a, b, epsi )`	$a \ge b$	$\|a-b\| \le \epsilon \ \vee \ (a-b) > 0$
`LMB_LE ( a, b, epsi )`	$a \le b$	$\|a-b\| \le \epsilon \ \vee \ (a-b) < 0$
`LMB_GT ( a, b, epsi )`	$a > b$	$\|a-b\| > \epsilon \ \wedge \ (a-b) > 0$
`LMB_LT ( a, b, epsi )`	$a < b$	$\|a-b\| > \epsilon \ \wedge \ (a-b) < 0$

Here `epsi` $\ge$ `0.D0` specifies an absolute tolerance. Set `epsi` negative to specify a relative tolerance which causes ϵ to be replaced by $\epsilon|a|$ or $\epsilon|b|$, whichever is larger. Note that in case $\epsilon = 0$ the functions just behave as FORTRAN relational operators.

4 Vector Operations

In MBUTIL there are a few routines to perform very elementary vector operations. In the following `a`, `b` and `c` are double precision arrays, dimensioned to at least `n` in the calling routine, and `n` is the dimension of the vectors stored in these arrays.

`SMB_VFILL ( a, n, val )`	Set all elements of `a` to `val`
`SMB_VMULT ( a, n, val )`	Multiply all elements of `a` by `val`
`SMB_VCOPY ( a, *b, n )`	Copy vector `a` to vector `b`
`SMB_VADDV ( a, b, *c, n )`	Compute `c` = `a` + `b` (`c` can be `a` or `b`)
`SMB_VMINV ( a, b, *c, n )`	Compute `c` = `a` − `b` (`c` can be `a` or `b`)
`DMB_VDOTV ( a, b, n )`	Compute the inproduct `a` · `b`
`DMB_VNORM ( m, a, n )`	Compute the norm $\|\mathtt{a}\|_{\mathtt{m}}$ (see below)
`LMB_VCOMP ( a, b, n, epsi )`	True if `a` == `b` within tolerance ϵ.

[3] Numbers in the underflow gap are too small to fit into a floating-point register and are set to zero.

The norms computed by the function `dmb_vnorm` are defined by

$$\|a\|_{m=0} = \max_i |a_i| \qquad \text{and} \qquad \|a\|_{m>0} = \left[\sum_i |a_i|^m \right]^{1/m} .$$

Thus we get the max norm for `m` = `0`, the city-block norm[4] for `m` = `1` and the Euclidean norm for `m` = `2`.

5 Triangular and Diagonal Band Equations

In QCDNUM there are lower triangular and lower diagonal band systems to be solved. For reasons of efficient storage and speed, we provide a set of routines optimised for lower and upper systems in different storage schemes. The naming convention of the lower (`L`) and upper (`U`) triangular equation solvers is

```
call SMB_LEQSx( arguments )          call SMB_UEQSx( arguments )
```

where the label `x` indicates the storage scheme (described below):

`M` =	Square matrix	`T` =	Packed triangular storage
`L` =	Linear storage	`B` =	Packed band storage
`V` =	Vector storage		

```
call SMB_LEQSM ( A, na, m, *x, b, n, *ierr )
```

Solve, by forward substitution, the lower diagonal band equation $\boldsymbol{Ax} = \boldsymbol{b}$ of dimension n and bandwidth m. When $n = m$, the equation is lower triangular.

`A`	Square array declared in the calling routine as `A(na,na)`. The $n \times n$ sub-matrix of `A` should be filled with the lower triangular or lower diagonal band matrix.
`na`	Dimension of `A` as declared in the calling routine.
`m`	Bandwidth `m` $\leq$ `n`.
`x`	Solution vector, dimensioned to at least `n` in the calling routine.
`b`	Right-hand side vector, dimensioned to at least `n` in the calling routine.
`n`	Dimension of the triangular system to be solved. When you set `n` negative, the routine computes the product $\boldsymbol{b} = \boldsymbol{Ax}$ for given input vector `x`.
`ierr`	Set to a non-zero value if A is singular. The solution vector is then undefined.

[4] Like the distance to your destination in a city with a rectangular street-grid.

```
call SMB_UEQSM ( A, na, m, *x, b, n, *ierr )
```

As above but now solve an upper triangular or upper diagonal band system.

For both routines the input matrix A is stored in a 2-dimensional FORTRAN array like

$$A_{ij}^{\mathrm{M}} = \begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ A_{31} & A_{32} & A_{33} & \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \qquad \text{or} \qquad A_{ij}^{\mathrm{M}} = \begin{pmatrix} A_{11} & A_{12} & & \\ & A_{22} & A_{23} & \\ & & A_{33} & A_{34} \\ & & & A_{44} \end{pmatrix}.$$

In this scheme $n \times n$ words of storage are used but only $n(n+1)/2$ words are occupied by a triangular matrix and even less by a band matrix. For better use of memory and CPU we provide a set of routines that employ more efficient storage and faster addressing schemes. In the descriptions below the matrices show the addresses assigned by a given scheme to the elements of a 4×4 matrix.

Linear storage (`L`): The address $k(i,j)$ in a linear column-wise store is given by

$$k(i,j) = i + (j-1)n \qquad k_{ij}^{\mathrm{L}} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}.$$

This storage model takes as much space as a normal FORTRAN array ($n \times n$ words) but the address arithmetic in the substitution loops is faster (factor of two, roughly).

Triangular storage (`T`): The storage model for lower triangular matrices is

$$k(i,j) = i(i-1)/2 + j \qquad (i \geq j) \qquad k_{ij}^{\mathrm{LT}} = \begin{pmatrix} 1 & & & \\ 2 & 3 & & \\ 4 & 5 & 6 & \\ 7 & 8 & 9 & 10 \end{pmatrix}$$

and for upper triangular matrices

$$k(i,j) = (n+1-i)(n-i)/2 + n + 1 - j \qquad (i \leq j) \qquad k_{ij}^{\mathrm{UT}} = \begin{pmatrix} 10 & 9 & 8 & 7 \\ & 6 & 5 & 4 \\ & & 3 & 2 \\ & & & 1 \end{pmatrix}.$$

Note that the address k is *not* linear in i but linear in j allowing for fast indexing in the forward or backward substitution loop. These storage schemes occupy $n(n+1)/2$ words and are fully efficient for triangular but not for band matrices.

Band storage (`B`): Storage of band matrices in $n \times m$ words is achieved by

$$k(i,j) = (i-j)n + i \qquad (i \geq j) \qquad k_{ij}^{\mathrm{LB}} = \begin{pmatrix} 1 & & & \\ 6 & 2 & & \\ & 7 & 3 & \\ & & 8 & 4 \end{pmatrix}$$

$$k(i,j) = (j-i)n + j \qquad (i \leq j) \qquad A_{ij}^{\mathrm{UB}} = \begin{pmatrix} 1 & 6 & & \\ & 2 & 7 & \\ & & 3 & 8 \\ & & & 4 \end{pmatrix}.$$

These schemes are not fully efficient since $m(m-1)/2$ words are wasted (word 5 in the example above) but they are better than the triangular schemes when $nm < n(n+1)/2$, that is, when the bandwidth $m < (n+1)/2$. Note that, again, the indexing is linear in j allowing for fast addressing in the substitution loop.

Vector storage (`V`): This is intended for Toeplitz triangular matrices for which the elements depend on the difference $i-j$. Thus it is sufficient to store only the first column (lower triangular) or row (upper triangular) of the matrix, taking m words of storage. All matrices in QCDNUM are of this kind. For lower triangular matrices we have

$$k(i,j) = i - j + 1 \qquad (i \geq j) \qquad k_{ij}^{\mathrm{LV}} = \begin{pmatrix} 1 & & & \\ 2 & 1 & & \\ 3 & 2 & 1 & \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

and for upper triangular matrices

$$k(i,j) = j - i + 1 \qquad (i \leq j) \qquad k_{ij}^{\mathrm{UV}} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ & 1 & 2 & 3 \\ & & 1 & 2 \\ & & & 1 \end{pmatrix}.$$

In the table below we list the routines to solve triangular or banded equations in the various storage schemes, and the functions to compute an address in the linear store `A`.

Equation solver	Address function	Size of `A`	Scheme
`SMB_LEQSL(A,m,x,b,n,ierr)`	`IMB_LADRL(i,j,m,n)`	$n \times n$	Linear
`SMB_UEQSL(A,m,x,b,n,ierr)`	`IMB_UADRL(i,j,m,n)`		
`SMB_LEQST(A,m,x,b,n,ierr)`	`IMB_LADRT(i,j,m,n)`	$n(n+1)/2$	Triangular
`SMB_UEQST(A,m,x,b,n,ierr)`	`IMB_UADRT(i,j,m,n)`		
`SMB_LEQSB(A,m,x,b,n,ierr)`	`IMB_LADRB(i,j,m,n)`	$n \times m$	Band
`SMB_UEQSB(A,m,x,b,n,ierr)`	`IMB_UADRB(i,j,m,n)`		
`SMB_LEQSV(A,m,x,b,n,ierr)`	`IMB_LADRV(i,j,m,n)`	m	Vector
`SMB_UEQSV(A,m,x,b,n,ierr)`	`IMB_UADRV(i,j,m,n)`		

Here `A` is a one-dimensional input array (the required size is given in the table above); `m` $\leq$ `n` is the bandwidth of the system; `x` is the output solution vector; `b` is the input right-hand side vector and `n` is the dimension of the system to be solved. The error flag `ierr` is set to a non-zero value if `A` is singular in which case the solution vector `x` is undefined. For negative `n` the product $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ is computed for a given input vector `x`.

The address functions return zero when (i,j) does not correspond to an element of the triangular or band matrix. This feature is exploited by the following code which packs a lower diagonal band of a square matrix `S` into a linear array `A` using, in this example, the triangular storage scheme.

```
      parameter (n=5,m=3)
      dimension S(n,n), A(n*(n+1)/2), x(n), b(n)

C--   Pack lower diagonal band in the triangular storage scheme
```

```
      do i = 1,n
        do j = 1,n
          k  = imb_LADRT(i,j,m,n)
          if(k.ne.0) A(k) = S(i,j)
        enddo
      enddo
C--   Solve lower band system
      call smb_LEQST(A,m,x,b,n,ierr)
```

6 Pointer Arithmetic in a Linear Store

In this section we describe a pointer arithmetic which maps multi-dimensional arrays onto linear storage so that it is possible to declare one large linear store at compilation time and dynamically partition it at run time. This simple method of dynamic memory management already provides many advantages compared to using fixed FORTRAN arrays: the QCDNUM dynamic memory, for instance, is based on calls to `smb_bkmat`.

To make clear how it works let us declare a 3-dimensional FORTRAN array

```
      dimension A ( i1min:i1plus, i2min:i2plus, i3min:i3plus )
```

The number of words occupied by A is given by $n_A = n_1 n_2 n_3$ with $n_k = i_k^+ - i_k^- + 1$. Instead of declaring A, we now partition a linear storage B which itself is declared as

```
      dimension B ( m1:m2 )
```

with $m_2 \geq m_1 + n_A - 1$ if B is to hold the data stored in A. It is easy to construct a linear pointer function $P(i_1, i_2, i_3)$ which assigns a unique address m to any possible combination of the indices:

$$m = P(i_1, i_2, i_3) = C_0 + C_1\, i_1 + C_2\, i_2 + C_3\, i_3. \tag{1}$$

The coefficients C_k are unique functions of $i_1^{\pm}$, $i_2^{\pm}$, $i_3^{\pm}$ and m_1, provided that a convention of 'row-wise' or 'column-wise' storage is adopted. We take in MBUTIL the FORTRAN column-wise convention where the first index 'runs fastest', that is:

$$P(i_1 + 1, i_2, i_3) \equiv P(i_1, i_2, i_3) + 1.$$

In the following we describe the routine `smb_bkmat` which defines the partition of the linear store (much like a FORTRAN dimension statement) and the function `imb_index` which calculates an address in this linear store.

```
call SMB_BKMAT ( imin, imax, *karr, n, im1, *im2 )
```

Define a partition of a linear store B such that it maps onto a multi-dimensional array $A(i_1, \ldots, i_n)$ with n indices. The definition range[5] of each index is $i_k^- \leq i_k \leq i_k^+$.

[5] An index i with identical lower and upper limits is a dummy index. The partition algorithm sets $C_i = 0$ for a dummy index so that an address does not depend on its value. A dummy thus simply acts as a placeholder in the list of indices and we may skip over them in calculating an address.

`imin` Input integer array containing the lower index limits i_k^-. Should be dimensioned to n in the calling routine.

`imax` As above, but now containing the upper limits i_k^+.

`karr` Integer array containing, on exit, the coefficients C_k used to calculate the address in the linear storage. Should be dimensioned to `karr(0:n)` in the calling routine.

`n` Dimension of the partition.

`im1` Address in B where the first word of A should be stored.

`im2` Gives, on exit, the address in B where the last word of A will be stored. B should thus be dimensioned to at least `B(im1:im2)`.

Note that once a partition is defined there is nothing against booking another one (with its own `karr`) starting at `im2+1`, provided that the store B is large enough.[6]

```
iaddr = IMB_INDEX ( iarr, karr, n )
```

Calculate an address in the linear store.

`iarr` Input integer array containing the values $(i_1, \ldots, i_n)$ of the indices.

`karr` Input integer array containing the coefficients C_k to calculate the address in the linear store, filled beforehand by a defining call to `smb_bkmat`.

`n` Dimension of the partition.

Note that this function does not perform array boundary checks so that you have to make sure that all indices stored in `iarr` are within their respective ranges.

Note also that `smb_bkmat` and `imb_index` do not *operate* on the store B but merely calculate an address in B. Thus you can book as many arrays as you want, and check after the final call to `smb_bkmat` that `im2` does not exceed the size of B.

The address arithmetic given in (1) provides the possibility to do fast addressing in nested loops. To see this, take for example a 3-dimensional array `A(100,100,100)` and map it onto a linear store `B(1000000)`. To calculate the addresses in `B` it is convenient to introduce—as an alternative to `imb_index`—the pointer function

```
P(i,j,k) = karr(0) + karr(1)*i + karr(2)*j + karr(3)*k
```

Now consider the loop:

```
do i = 1,100
  do j = 1,100
    do k = 1,100
      Aijk = B( P(i,j,k) )    !A(i,j,k)
      ..
```

[6] It is a good idea to offset the storage by $n+3$ words and use these words to store `im2+1` (pointer to next partition), `n`, and `karr`. In this way a linear store carries metadata describing its own structure.

The address calculation in the inner loop costs 3×10^6 additions and 3×10^6 multiplications. However, from (1) it follows that increasing or decreasing an index by one unit corresponds to a unique shift of the address in B. Fast addressing can then be achieved by maintaining running sums of these shifts, as shown below.

```
ia     = P(1,1,1)             !Start address
ishift = P(2,1,1) - ia        !Address shift of i
jshift = P(1,2,1) - ia        !Address shift of j
kshift = P(1,1,2) - ia        !Address shift of k

do i = 1,100
  ja = ia
  do j = 1,100
    ka = ja
    do k = 1,100
       Aijk = B(ka)           !A(i,j,k)
       ka   = ka + kshift
    enddo
    ja = ja + jshift
  enddo
  ia = ia + ishift
enddo
```

There are now slightly more than 10^6 additions (no multiplications) to calculate the addresses. Note that this addressing scheme works for any nesting order of the loops. When the nesting is in the same order as the indices, with the first index running in the inner loop, then one walks sequentially through the store and the code simplifies to:

```
ia = P(1,1,1)          !Start address
do k = 1,100
  do j = 1,100
    do i = 1,100
      Aijk = B(ia)     !A(i,j,k)
      ia   = ia + 1
      ..
```

Here is code with very fast addressing that initialises the array

```
ia = P(  1,  1,  1)    !First address
ib = P(100,100,100)    !Last  address
do i = ia,ib
  B(i) = value
enddo
```

7 Fast Interpolation

Piecewise polynomial interpolation of order n on tabulated data consists of selecting an n-point sub-grid (mesh) around the interpolation point, followed by an interpolation of the data on that mesh. This interpolation is computed as a (nested) weighted sum

with weights that depend on the interpolation point but not on the data. Pre-calculating these weights thus allows for fast interpolation, at a given point, of more than one table.

We will restrict ourselves here, as in QCDNUM, to the orders $n = 1$ (point value), 2 (linear interpolation) and 3 (quadratic interpolation). The interpolation meshes of size $n = (1, 2, 3)$ are shown in Figure 1. For these meshes, the algorithm reads

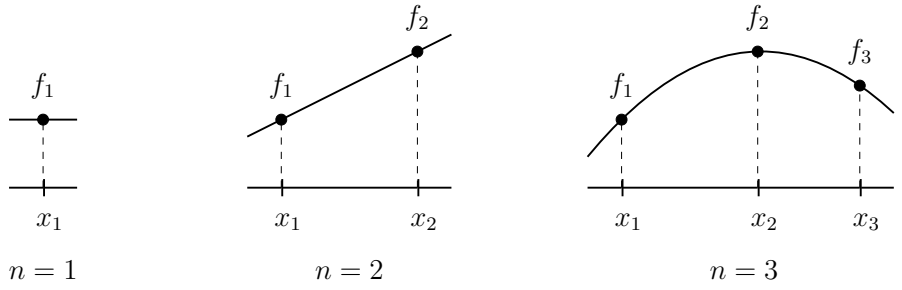

Figure 1: Interpolation function for meshes of size $n = 1$, 2 and 3.

$n = 1$	$f(x) = w_1(x)\, f_1$	$w_1(x) = 1$	
$n = 2$	$f(x) = w_1(x)\, f_1 + w_2(x)\, f_2$	$w_1(x) = (x_2 - x)/(x_2 - x_1)$	$w_2 = 1 - w_1$
$n = 3$	$g(x) = w_1(x)\, f_1 + w_2(x)\, f_2$	$w_1(x) = (x_2 - x)/(x_2 - x_1)$	$w_2 = 1 - w_1$
	$h(x) = w_3(x)\, f_2 + w_4(x)\, f_3$	$w_3(x) = (x_3 - x)/(x_3 - x_2)$	$w_4 = 1 - w_3$
	$f(x) = w_5(x)\, g \ + w_6(x)\, h$	$w_5(x) = (x_3 - x)/(x_3 - x_1)$	$w_6 = 1 - w_5$.

It is seen that the interpolation is, for a given interpolation point x, a (nested) weighted sum of the function values f_i, with $(1, 2, 6)$ different weights for interpolation at order $n = (1, 2, 3)$. The weights depend on x and the n mesh points x_i, but not on f.

For a 2-dimensional interpolation on an $n_x \times n_y$ mesh we simply perform n_y interpolations in x—here a pre-calculation of weights already pays-off—and one interpolation in y.

When we have to interpolate more than one table it clearly makes sense to first calculate the weights and then interpolate each table. For this we provide the routine `smb_polwgt` to pre-compute the weights which can then be fed into the interpolation functions `dmb_polin1` and `dmb_polin2` for 1- or 2-dimensional interpolation, respectively.

```
call SMB_POLWGT ( x, xi, n, *w )
```

Compute the weights for interpolation on a $1, 2, 3$-point interpolation mesh.

- `x` Interpolation point (irrelevant when `n` = 1). Should be inside the range of the interpolation mesh to avoid extrapolation and the corresponding loss of accuracy.
- `xi` Input array, dimensioned to at least `n` in the calling routine, filled with the `n` interpolation mesh points x_i (see Figure 1).
- `n` Number of points in the interpolation mesh (interpolation order) [1–3].
- `w` Output weight array, dimensioned to at least (1,2,6) for `n` = (1,2,3).

```
val = DMB_POLIN1 ( w, fi, n )
```

One-dimensional interpolation on a $1, 2, 3$-point interpolation mesh.

`w` Input weight array filled by an upstream call to `smb_polwgt`.

`fi` Input array, dimensioned to at least `n` in the calling routine, filled with `n` function values f_i (see Figure 1).

`n` Interpolation order [1–3] as set in the upstream call to `smb_polwgt`.

```
val = DMB_POLIN2 ( wx, nx, wy, ny, fij, m )
```

Two-dimensional interpolation on an $n_x \times n_y$ interpolation mesh.

`wx` Input weight array filled by an upstream call to `smb_polwgt`.

`nx` Interpolation order in x [1–3] as set in the upstream call to `smb_polwgt`.

`wy`, `ny` As above, but now for the interpolation in y.

`fij` Input array, dimensioned to at least `fij(nx,ny)` in the calling routine, filled with the function values f_{ij} to be interpolated.

`m` First dimension of `fij` as declared in the calling routine.

In the example below we interpolate three functions on a 3×2 interpolation mesh.

```
dimension xi(3), wx(6), yi(2), wy(2), fij(3,2), gij(3,2), hij(3,2)
..
fill the arrays xi, yi and fij, gij, hij (code not shown)
..
call smb_polwgt( x, xi, 3, wx )          ! weights in x
call smb_polwgt( y, yi, 2, wy )          ! weights in y
f  = dmb_polin2( wx, 3, wy, 2, fij, 3 )  ! f(x,y)
g  = dmb_polin2( wx, 3, wy, 2, gij, 3 )  ! g(x,y)
h  = dmb_polin2( wx, 3, wy, 2, hij, 3 )  ! h(x,y)
```

8 Bitwise Operations

In this section we describe a few routines and functions to manipulate bits in 32-bit integers. The bit numbering runs from 1 (least significant bit) to 32 (most significant bit). The routines presented below rely on the following machine representation of the integer value +1 which, as far as we know, is standard on all platforms:

```
bit 32                              bit 1
  |                                   |
  00000000000000000000000000000001
```

Please notice that the bitwise operations will *not* work for 16-bit integers.

```
call SMB_SBIT1 ( i, n )
```

Set bit n of integer i to 1.

```
call SMB_SBIT0 ( i, n )
```

Set bit n of integer i to 0.

```
ival = IMB_GBITN ( i, n )
```

Give the value (0 or 1) of bit n of integer i.

```
ival = IMB_SBITS ( cpatt )
```

Store a bit pattern in an integer i. The bit pattern is given in the input string `cpatt` containing a sequence of 32 characters '1' and '0'.

Here is an example which sets the value of i to 7:

```
character*32 cpatt
data cpatt /'00000000000000000000000000000111'/

i = imb_sbits ( cpatt )      ! i = 7
```

```
call SMB_GBITS ( i, *cpatt )
```

Store the bit pattern of an integer i in a character string. The output string `cpatt` should be declared `character*32` in the calling routine.

```
ierr = IMB_TEST0 ( mask, i )
```

Verify that a selected set of bits in i are all set to zero.

`mask` Input 32-bit integer. If bit n of `mask` is set to 1 then the corresponding bit of i will be checked. Otherwise bit n will not be checked.

`i` Input 32-bit integer variable to be checked.

`ierr` Non-zero if the test fails. Thus `ierr` = `0` means that all checked bits in i are 0.

```
ierr = IMB_TEST1 ( mask, i )
```

Verify that a selected set of bits in i are all set to one.

`mask` Input 32-bit integer. If bit n of `mask` is set to 1 then the corresponding bit of i will be checked. Otherwise bit n will not be checked.

`i` Input 32-bit integer variable to be checked.

`ierr` Non-zero if the test fails. Thus `ierr` = `0` means that all checked bits in i are 1.

9 Character String Manipulations

In this section we describe a few routines which perform elementary character string manipulations. It is recommended to explicitly initialise strings to a series of blank characters at program start-up. This is easily done by using `smb_cfill`.

```
call SMB_CFILL ( char, string )
```

Fill the character variable `string` with the character `char`.

`char` Input one-character string.

`string` Character string declared `character*n` in the calling routine. On exit all n characters of `string` will be set to `char`.

```
call SMB_CLEFT ( string )
```

Left adjust the characters in `string`, padding blanks to the right.

```
call SMB_CRGHT ( string )
```

Right adjust the characters in `string`, padding blanks to the left.

```
call SMB_CUTOL ( string )
```

Convert the character variable `string` to lower case.

```
call SMB_CLTOU ( string )
```

Convert the character variable `string` to upper case.

```
ipos = IMB_LASTC ( string )
```

Returns the position of the rightmost non-blank character in `string` (`0` for empty strings). This function measures the actual length of a string unlike the FORTRAN function `len()` which returns for a `character*n` variable the number `n`.

```
ipos = IMB_FRSTC ( string )
```

Returns the position of the leftmost non-blank character in `string` (`0` for empty strings).

```
lvar = LMB_COMPS ( stringa, stringb, istrip )
```

Case-independent comparison of two character strings. Trailing blanks are stripped-off and leading blanks when you set `istrip` = `1`. Both `lvar` and `lmb_comps` should be declared logical in the calling routine.

```
lvar = LMB_MATCH ( string, substr, cwild )
```

Verify that the character string `substr` is contained in `string`. The string `substr` may, or may not, contain a wild character `cwild` which will match any character in `string`. The matching is case insensitive. Before processing, both `string` and `substr` have leading and trailing blanks stripped-off. Both `lvar` and `lmb_match` should be declared logical in the calling routine.

`string`	Non-empty input character string.
`substr`	Non-empty input character string that fits into `string` (after stripping).
`cwild`	Input character (wild character acting as placeholder).
`lvar`	Set to true if `substr` is contained in `string` and to false if it is not. Also set to false if, for some reason, the comparison cannot be done.

Here are a few examples:

```
lvar = lmb_match ( 'Amsterdam', ' am ', '*' )  !  .true.
lvar = lmb_match ( 'Amsterdam', '*am ', '*' )  !  .true.
lvar = lmb_match ( 'Amsterdam', ' am*', '*' )  !  .true.
lvar = lmb_match ( 'Amsterdam', '*am*', '*' )  !  .false.
```

```
call SMB_ITOCH ( ival, *chout, *lengout )
```

Convert an integer to a character string.

`ival`	Input integer.
`chout`	Character string containing, on exit, the digits of `ival`. Should be declared `character*n` in the calling routine. If `n` is smaller than the number of digits of `ival`, the string will be filled with asterisks (`*`).
`lengout`	Number of characters encoded in `chout`.

With this routine you can nicely embed integers into text strings as shown below:

```
character*10 string
ierr = -12
call smb_itoch(ierr,string,leng)   !unformatted write comes next
write(lun,*) 'Error ',string(1:leng),' encountered'
jerr =  12345
call smb_itoch(jerr,string,leng)   !formatted write comes next
write(lun,'(''Error '',A,'' encountered'')') string(1:leng)
```

This code will produce the strings (note the snug fit of the numbers)

```
Error -12 encountered
Error 12345 encountered
```

10 String Formatter

A powerful feature of FORTRAN is the use of strings as an internal file. A restriction is, however, that the FORTRAN77 standard does not allow list-directed (= free-format) read from internal files. For this reason we provide the routine `smb_sfmat` that determines the format of an arbitrary string so that one can do a *formatted* read on that string.

```
call SMB_SFMAT ( stin, *stout, *fmt, *ierr )
```

Bring a free-format input string into a standard format and give the FORTRAN format descriptor of the reformatted string.

`stin` Input character string. This string will be parsed into words and each word will be classified and reformatted (if necessary) as is described below.

`stout` Output character string with the (reformatted) input words. Must be declared `character*n` in the calling routine, with `n` large enough to hold the reformatted input string (error condition if `n` too small).

`fmt` Output character string with the FORTRAN format descriptor of `stout`. Should be declared `character*m` in the calling routine, with `m` large enough to hold the format descriptor (error condition if `m` too small).

`ierr` Output error flag (note that `stout` and `fmt` will be undefined upon error):

- `0` All OK.
- `1` Empty input string.
- `2` Found unbalanced quotes in `stin`.
- `3` Word count exceeded (max 100 words per string).
- `4` Word length exceeded (max 120 characters per word).
- `5` Not enough space in `stout`.
- `6` Not enough space in `fmt`.

The input string `stin` is parsed into words (these are defined as substrings separated by one or more blanks) and each word is classified as follows.

`L` Logical. A single `T` or `F` in upper case;

`I` Integer. Any signed or unsigned string of digits. Examples: `12`, `-12`, `+12`;

`F` Floating point number. Any signed or unsigned string of digits with a leading, embedded or trailing decimal point. Examples: `-.12`, `12.`, `1.2`. Note that the first two numbers will be reformatted to `-0.12` and `12.0`, respectively;

`E` Floating point number in exponential format. This is the character `E` or `e` preceded by a signed or unsigned integer or floating point number (mantissa) and followed by a signed or unsigned integer (exponent). Examples: `.12E1`, `1.2E0`, `12.E-1`, `12e-1`. These will all be reformatted to `0.12E1`;

`D` As above, but now in `D`-format;

`Q` Quoted string. Anything embedded in double single quotes, like `''foo bar''`;[7]

`A` Character string. Any word that is not classified as `L`, `I`, `F`, `E`, `D` or `Q`.

The use of the string formatter is illustrated by the datacard processor presented below. Suppose that we want to steer a program with datacards.

```
MYSUB   .3
MYSUB   2D-4  12.
```

The first card should call `mysub(par1,par2)` with a default value for `par2` while the second card should call that same subroutine but with both parameters taken from the card. From this example we see two desirable features of a datacard processor: (i) handling of variable length parameter lists[8] and (ii) handling of free-format input.

The first requirement can be fulfilled by first trying to read the datacard with two parameters and then, upon error, read it again with one parameter. The second requirement can in principle be fulfilled by a FORTRAN list-directed read.

```
character*5 key
read(unit=lun,fmt=*,err=100,end=100) key, par1, par2
```

Reading the first card in this way indeed produces an error, but not because the parameter list is exhausted but because the list-directed read skips to the next record and tries to read the character string `MYSUB` into the floating point variable `par2`. This brings us to the third requirement that (iii) only one card should be read at the time. This cannot be done with a list-directed read because it may read more than one record to fulfil the parameter list.

A solution to this is to read the datacard as a character string. Thus we enclose the cards in quotes (note that the alignment of the quotes below is irrelevant)

```
'  MYSUB   .3        '
'  MYSUB   2D-4  12. '
```

and code the card reading loop as

```
character*120 dcard
idum = 0
do while(idum.eq.0)
```

[7] A literal string in FORTRAN is embedded in single quotes: `foo` → `'foo'`. Quotes inside the string are represented by double quotes, thus: `foo's` → `'foo''s'`. To process quoted strings, the formatter looks for opening quotes which are double quotes at the beginning of a string, or those preceded by a blank. When opening quotes are detected, all following characters are classified as quoted string (`Q`) until closing quotes are found (error condition if not). Closing quotes are double quotes followed by a blank, or those at the end of the string. Quotes inside (quoted) strings are allowed but one should take care that the formatter does not interpret them as opening or closing quotes. In any case, if you use quoted strings it is always a good idea to print the output string `stout` and the format descriptor `fmt` to check that the formatter does what is intended.

[8] With the restriction that optional parameters should be put at the end of the list.

```
        read(unit=lun,fmt='(A)',err=100,end=100) dcard
      enddo
100   continue
```

This code clearly does a card-by-card reading of a datacard file. It also restricts the length of a card to 120 characters[9] but this is not much of a problem since longer cards tend to violate a fourth requirement: (iv) a datacard must be easy to read on a terminal.

After reading a card, the key (here with a length of 5 characters) can be separated from the parameter list by some straight-forward character string manipulation.[10]

```
      character*120 dcard, upars
      character*5   key5
      i1    = imb_frstc(dcard)
      key5  = dcard(i1:i1+4)
      upars = dcard(i1+5:)
```

Now we can use the string `upars` as an internal file and fetch the parameters from that string by a list-directed read.

```
      read(unit=upars,fmt=*,end=10,err=10) par1, par2
```

This probably works on your platform but list-directed read from strings is not supported by the FORTRAN77 standard so that this call is not guaranteed to be portable.

A solution is provided by calling the routine `smb_sfmat` that takes as an input any character string (`upars`, say) and produces a re-formatted output string (`fpars`), together with a format descriptor (`fmt`), thus:

```
      call smb_sfmat( upars, fpars, fmt, ierr )
```

For the example cards above this gives:

key	upars	fpars	fmt
MYSUB	.3	0.3	(1X,F3.1)
MYSUB	2D-4 12.	0.2D-3 12.0	(1X,D6.1,1X,F4.1)

Now we can fetch the parameters by a *formatted* read as is required by the FORTRAN77 standard.

```
      character*120 upars, fpars, fmt
      call smb_sfmat( upars, fpars, fmt, ierr )
      read(unit=fpars,fmt=fmt,end=10,err=10) par1, par2
```

Here are examples of keys that have a quoted string as a parameter:

[9] Cards of less than 120 characters are blank padded while longer cards are just truncated.

[10] To keep things simple we do not handle here empty strings, strings with only one word (no parameter list) and strings where the first word is not 5 characters long.

```
      ' LOGIC   T  ''F'' '
      ' VERSN ''Version 3.6 12-AUG-2014'' '
      ' NAMES   P.A.M.  Dirac  ''P.A.M. Dirac'' '
```

The first card has one logic (`L1`) and one character parameter (`A1`). The numbers in the second card are not classified as such since they are part of a quoted string (`A23`). The last card has three parameters: initials (`A6`), last name (`A5`) and full name (`A12`) which must be quoted because it contains an embedded blank.

Note that in some exceptional cases the format descriptor can become quite long as in

```
      stout = ' X X X X '    fmt = '( 1X,A1,1X,A1,1X,A1,1X,A1 )'
```

where each item occupies 2 characters in `stout`, but 6 characters in `fmt`.[11]

Putting it all together, we arrive at the card reading code shown below.

```
      subroutine cardread(lun)
      character*120 dcard, fpars, fmt
      character*5   key5                   !assume 5-character key

      idum = 0
      do while(idum.eq.0)
C--     The read statement will terminate the while-loop
        read(unit=lun,fmt='(A)',err=200,end=100) dcard
        i1    = imb_frstc(dcard)
        key5  = dcard(i1:i1+4)

        call smb_sfmat(dcard(i1+5:), fpars, fmt, ierr)

        if( key5 .eq. 'MYSUB' ) then
          read(unit=fpars,fmt=fmt,end=10,err=10) par1, par2
          call MYSUB(par1,par2)
          return
 10       read(unit=fpars,fmt=fmt,end=20,err=20) par1
          par2 = default
          call MYSUB(par1,par2)
          return
 20       stop 'Error reading MYSUB datacard'
        elseif( key5 .eq. ... ) then
          ..
          process other keys, if any
          ..
        else
          stop 'Unknown card'
        endif
      enddo

 100  return
 200  stop 'Error reading input file'
      end
```

[11] The formatter is not smart enough to generate repeat counts and write `fmt = '( 4(1X,A1) )'`.

Index