A Portable Collection of Fortran Utility Routines

MBUTIL version 2021-09-07

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

January 8, 2022

Abstract

The MBUTIL library is a well documented collection of FORTRAN routines—some of these with a C++ interface. The routines are are taken from the public libraries CERNLIB and NETLIB, or 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.

*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].

Several FORTRAN routines have C++ wrappers, as will be indicated in the margin of the page. All routines reside in the namespace `MBUTIL` and the routine names are written in lower case, unless otherwise stated. For further details see `arXiv:1602.08383`, or the QCDNUM write-up. C++

The syntax of the MBUTIL calls is as follows

```
call xMB_NAME ( arguments )          MBUTIL::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 )
```

Floating-point variables are in double precision, unless otherwise stated. This implies that, in FORTRAN, floating-point numbers must be passed in double precision format.[3]

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

The following routine returns the MBUTIL version number.

```
iver = IMB_VERSION( )          int iver = MBUTIL::imb_version();
```

2 Utility Routines

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

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

Calculate the gamma function

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

[1] `https://www.nikhef.nl/~h24/qcdnum`

[2] `http://www.netlib.org`

[3] This does not apply to calls in C++ that automatically does the conversion to `double`, if needed.

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
F012	`SMB_DSEQN ( n, a, m, *ierr, k, !b)`	Symmetric equations
	`SMB_TDIAG ( n, a, b, c, !d, *ierr)`	Tri-diagonal equations
M103	`SMB_RSORT ( !rarr, n )`	Sort real array
	`SMB_ASORT ( !rarr, n, *m )`	Sort and weed real array
	`RMB_URAND ( !iy )`	Uniform random numbers
	`IMB_IHASH ( ih, imsg, n )`	Hash (integer)
	`IMB_JHASH ( ih, dmsg, n )`	Hash (double → integer)
	`IMB_DHASH ( ih, dmsg, n )`	Hash (double)
	`IMB_NEXTL ( lmin )`	Find next free LUN

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 )
```
C++

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.

Also available are the fixed-point Gauss routines `dmb_gaus1|2|3|4(fun,a,b)` that need only (1,2,3,4) function evaluations and are exact up to a polynomial degree of $(1, 3, 5, 7)$.

The C++ interface requires that functions which are passed to routines have their arguments passed as pointers. Therefore the C++ prototype of `fun` should be: `double fun(double *x)`. C++

```
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_DSEQN ( n, arr, idim, *ierr, k, !b )
```

Solve the matrix equation $Ax = b$ with multiple right-hand sides. The matrix A must be symmetric positive definite (covariance matrix). Taken from CERNLIB F012 (`dseqn`).

`n` Number of equations to solve.

`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`.

`ierr` Set to 0 if A is found to be positive definite, to -1 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_TDIAG ( n, a, b, c, !d, *ierr )
```

Solve the tri-diagonal matrix equation $Ax = d$ which reads, in components,

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

`n` Number of equations to solve.

`a`, `b`, `c` Double precision arrays, containing the diagonal elements as shown above. The input values of `a(1)` and `c(n)` are ignored. On exit, `b` is destroyed.

`d` Double precision array that containins on entry the right-hand side vector d and on exit the solution vector x.

`ierr` Set to -1 if the tri-diagonal matrix is (close to) singular, to 0 otherwise.

If you set `a(2)` or `c(1)` to zero then A is taken to be upper or lower bi-diagonal; diagonal if both are zero. Uni- bi- and tri-diagonal matrix equations show up in the construction of linear, quadratic and cubic interpolation splines, for instance.

The tri-diagonal equations are solved with the so-called Thomas algorithm using code taken from `wikipedia.org`. The solution may be unstable; in case of problems, resort to other methods like Gauss elimination with pivoting. The Thomas algorithm is fast, O(n), and stable for diagonally dominant or positive definite symmetric matrices (covariance matrices). The bi-diagonal equations are solved by substitution.

```
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.

C++
```
ihash = IMB_IHASH ( ih, imsg, n )   ihash = IMB_J|DHASH ( ih, dmsg, n )
```

Produce a 4-byte hash of a list of `n` numbers (message). The input can be an integer array (`imb_ihash`), a double precision array with type-converted integers (`imb_jhash`), or a double precision array with floating-point numbers (`imb_dhash`).[4]

The seed `ih` should initially be set to zero. If more than one call is made to hash a message (*e.g.* a multi-format message distributed over several arrays) then `ih` should be set to zero for the first call and to the previous hash for the next calls to `imb_ihash`, `jhash` or `dhash`. Here is an example where integers are processed one-by-one:

```
ih  = 0                                 int ih = 0;
do i = 1,n                              for(int i=0; i<n; i++){
  ih = IMB_IHASH(ih,imsg(i),1)            ih = MBUTIL::imb_ihash(ih,imsg+i,1);
enddo                                   }
```

Proper seeding guarantees that the hash only depends on the message and not on how it is processed. We use the Pearson hash-function with code taken from `wikipedia.org`.

```
lun = IMB_NEXTL ( lmin )
```

Find a free logical unit number above `lun` = `10` or `lmin`, whichever is larger.

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.[5] 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 compare double precision numbers within a tolerance `epsi`.

[4]Before making the hash, `imb_jhash` does a double-to-integer conversion whereas `imb_dhash` decomposes each number into an integer mantissa, rounded to 9 digits, and an integer exponent. The floating-point precision is restricted to 9 digits because the mantissa has to fit into an integer.

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

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 )`	$a < b$	$\|a-b\| > \epsilon \ \wedge \ (a-b) < 0$

C++

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 Bitwise Operations

In this section we describe a few routines and functions to manipulate bits in 32-bit integers. The bits are numbered from 1 (least significant bit, LSB) to 32 (most significant bit, MSB) and are grouped into four bytes, as shown below.

32 ④ 25	24 ③ 17	16 ② 9	8 ① 1

Bitwise operations are not part of standard FORTRAN77 but intrinsic functions exist that are supported by most, if not all, modern compilers. However, the bit-index may then range from 0–31 instead of 1–32, as is our convention. The MBUTIL bitwise routines therefore only use intrinsic functions that do not depend on the bit-index.

Intrinsic function	Description
`i1 = NOT(i2)`	Bitwise negation
`i1 = IAND(i2,i3)`	Bitwise and
`i1 = IOR(i2,i3)`	Bitwise inclusive or
`i1 = IEOR(i2,i3)`	Bitwise exclusive or
`i1 = ISHFT(i2,n)`	Left-shift (`n` $>$ `0`) or right-shift (`n` $<$ `0`)

Below are higher-level bitwise routines derived from the intrinsic functions listed above.

```
call SMB_SBIT1 ( i, n )          call SMB_SBIT0 ( i, n )
```

Set bit `n` of integer `i` to `1` or to `0`. When `n` is outside the range [`1`–`32`] then set all bits.

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

Give the value of bit `n` of integer `i`. Returns `-1` when `n` is out of range [`1`–`32`].

```
call SMB_CBYTE ( i1, ibyte1, i2, ibyte2 )
```

Copy `ibyte1` of `i1` to `ibyte2` of `i2`. Do-nothing when `ibyte` is out of range [`1`–`4`].

```
i = IMB_SBITS ( bpatt )        call SMB_GBITS ( i, *bpatt )
```

Store or retrieve the bits of an integer `i`. The bit pattern is encoded—left adjusted—in the string `bpatt` which must be at least of size `character*32`. Here is an example.

```
character*35 bpatt
data bpatt /'00000000000000000000000000000111   '/  !left adjusted

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

```
call SMB_BYTES ( bpatt, *bytes )
```

Insert blanks at the byte-boundaries of the bit-string `bpatt`. The output string `bytes` must be at least of size `character*35` and can be the same string as the input `bpatt`.

```
01001001100101100000001011010010  becomes  01001001 10010110 00000010 11010010
```

```
ierr = IMB_TEST0 ( mask, i )     ierr = IMB_TEST1 ( mask, i )
```

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

5 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 )      call SMB_CRGHT ( string )
```

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

```
call SMB_CUTOL ( string )      call SMB_CLTOU ( string )
```

Convert the character variable `string` to lower (`UTOL`) or upper case (`LTOU`).

```
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 made.

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.
```

C++ `call SMB_ITOCH ( ival, *chout, *leng )`

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 (`*`).
`leng`	Number of characters encoded in `chout`.

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

```
character*10 ctemp
character*80 text
itemp = -273
call smb_itoch(itemp,ctemp,n)
text = 'The absolute zero is '//ctemp(1:n)//' degree Celsius'
```

This code will produce the string (note the nice fit of the number)

```
The absolute zero is -273 degree Celsius
```

C++ `call SMB_DTOCH ( dval, n, *chout, *leng )`

As `smb_itoch` but now tightly format a double precision number. The input argument `n` specifies how many digits are to be kept in the mantissa of `dval` [1–9].

The example below shows how `n` influences the formatting. Numbers are rounded, not truncated, and trailing zero's behind a decimal point are discarded. A number is displayed in integer, floating point or exponential format, whatever fits best.

```
2345.0D0   with 5 digits   →   2345
2345.6D0   with 5 digits   →   2345.6
2345.6D0   with 4 digits   →   2346.
2345.6D0   with 3 digits   →   2.35E+03
```

The following C++ example shows how the routines `smb_itoch` and `smb_dtoch` can nicely take care of formatting integers and doubles. Note that the output argument `leng` is not used in C++.

C++
```
string chout; int leng;
MBUTIL::smb_itoch(-273,chout,leng);
cout << "The absolute zero is " << chout << " degree Celsius" << endl;
MBUTIL::smb_dtoch(2345.6,3,chout,leng);
cout << " 2345.6 with 3 digits --> " << chout << endl;
```

```
call SMB_HCODE ( ihash, *hcode )
```
C++

Extract the four byte-values of `ihash` and put them in the string `hcode` (hash code) which should be declared at least `character*15` in the calling routine. For example,

```
hash integer 1234567890 becomes hash code 073-150-002-210
```

```
int imsg[n]; string hcode;
int ihash = MBUTIL::imb_ihash(0, imsg, n);
MBUTIL::smb_hcode(ihash, hcode);
```
C++

6 Vector Operations

In the following routines `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_IFILL ( ia, n, ival )`	Set all elements of `ia` to `ival`
`SMB_VMULT ( a, n, val )`	Multiply all elements of `a` by `val`
`SMB_VCOPY ( a, *b, n )`	Copy vector `a` to vector `b`
`SMB_ICOPY ( ia, *ib, n )`	Copy vector `ia` to vector `ib`
`SMB_VITOD ( ia, *b, n )`	Copy with integer to double conversion
`SMB_VDTOI ( a, *ib, n )`	Copy with double to integer conversion
`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
`DMB_VPSUM ( a, w, n )`	Pairwise summation of elements `a(i)`

C++

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[6] for `m` = `1` and the Euclidean norm for `m` = `2`.

The function `dmb_vpsum` recursively sums pairs of terms until all `n` elements of an array `a` are summed.

```
sum = dmb_vpsum(a,w,n)
```

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

Here `w` is a working array that should be dimensioned in the calling routine to at least `n+1`. The idea behind pairwise addition is that only numbers of comparable magnitude are added, instead of adding terms to a large running sum and losing numerical precision in the process. However, accuracy is rarely a problem when the summation is carried out in double precision.

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 for $n = (1, 2, 3)$ are shown in Figure 1.

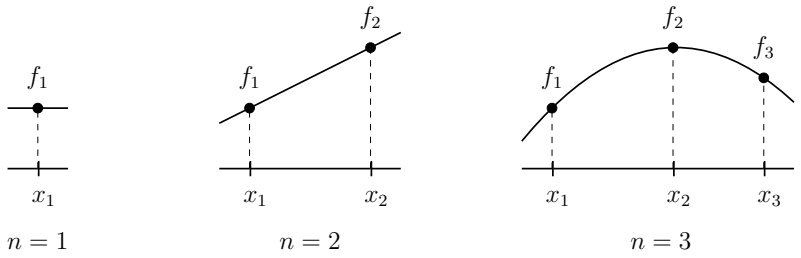

Figure 1: Interpolation function for meshes of size $n = 1$, 2 and 3.

For these meshes, the (Neville's) algorithm reads

$$
\begin{array}{lllll}
n=1 & f(x) = w_1(x)\, f_1 & w_1(x) = 1 & & \\
n=2 & f(x) = w_1(x)\, f_1 + w_2(x)\, f_2 & w_1(x) = (x_2 - x)/(x_2 - x_1) & w_2 = 1 - w_1 & \\
n=3 & g(x) = w_1(x)\, f_1 + w_2(x)\, f_2 & w_1(x) = (x_2 - x)/(x_2 - x_1) & w_2 = 1 - w_1 & \\
 & h(x) = w_3(x)\, f_2 + w_4(x)\, f_3 & w_3(x) = (x_3 - x)/(x_3 - x_2) & w_4 = 1 - w_3 & \\
 & f(x) = w_5(x)\, g \;\; + w_6(x)\, h & w_5(x) = (x_3 - x)/(x_3 - x_1) & w_6 = 1 - w_5. &
\end{array}
$$

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 Cubic spline interpolation

A cubic spline is a set of piecewise third-degree polynomials defined on intervals in x:

$$S(x_i \le x < x_{i+1}) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \qquad (1)$$

At the node-points x_i, S is continuous up to the third derivative, which is allowed to be discontinuous. To uniquely define the spline for a given n-point interpolation, two boundary conditions are usually imposed on the derivatives $S'''(x_1)$ and $S'''(x_n)$.

An algorithm to compute the coefficients b, c and d is widely available on the web.[7] In this algorithm the third derivatives at x_1 and x_n are estimated from divided differences.

C++
```
SMB_SPLINE( n, x, y, *b, *c, *d )
```

Compute the coefficients b_i, c_i and d_i of a cubic interpolation spline through $n \ge 2$ points (x_i, y_i), as defined by (1). Here `x`, `y`, `b`, `c` and `d` are double precision arrays dimensioned to at least `n` in the calling routine. On entry the array `x` must be filled with `n` interpolation points x_i in strictly ascending order, and `y` with the ordinates y_i. If `n` = 2 or 3 the routine returns a simple linear or quadratic interpolation polynomial.

C++
```
val = DMB_SEVAL( n, u, x, y, b, c, d )
```

Compute the interpolation spline $S(u)$ defined by an upstream call to `smb_spline`. Returns an extrapolation in case $u < x_1$ or $u > x_n$.

C++ The C++ prototypes are, without the scope resolution operator `MBUTIL::`,

```
void         smb_spline(int n, double* x, double* y, ... , double* d)
double val = dmb_seval( int n, double  u, double* x, ... , double* d)
```

9 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 memory manager WSTORE, for instance, is based on `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

[7] As far as we know, the code in MBUTIL originates from G.E. Forsythe, M.A. Malcolm and C.B. Woler, '*Computer Methods for Mathematical Computations*' (Prentice-Hall, 1977).

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

with $m_2 \geq m_1 + n_A - 1$ if B is to contain the array 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:

$$m = P(i_1, i_2, i_3) = C_0 + C_1\, i_1 + C_2\, i_2 + C_3\, i_3. \tag{2}$$

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[8] of each index is $i_k^- \leq i_k \leq i_k^+$.

`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^+, with $i_k^+ \geq i_k^-$.

`karr` Integer array containing, on exit, the coefficients C_k used to calculate the address in the linear storage. Should be dimensioned to at least $n+1$ in the calling routine. In the following we assume that it is declared as `karr(0:n)`.

`n` Dimension of the partition.

`im1` Address in B where the first word of A should be stored.[9]

`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.

```
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.

[8] 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.

[9] 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.

`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 (2) 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)
      ..
```

The address calculation in the inner loop costs 3×10^6 additions and 3×10^6 multiplications. However, from (2) 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
```

Index