The Solvers in BSPLINES

Trach-Minh Tran v0.6, December 2011

Implementation of a common simple interface to popular solver packages (LAPACK, PARDISO, WSMP, PETSc, etc.). The main goal is to provide an easy access to these packages in order to solve elliptic and parabolic as well as some types of integro-differential equations.

Contents

1 Matrix form of the problem

1.1 Getting started

1.1.1 A one-dimensional problem

Let us start with the one-dimensional Sturm-Liouville differential equation:

$$-\frac{d}{dx}\left[C_1(x)\frac{d\phi}{dx}\right] + C_2(x)\phi = \rho,$$

on the domain $0 \leq x \leq L$ with suitable boundary conditions. On a grid with N intervals, the discretized solution ϕ, using the splines $\Lambda_i(x)$ of order p can be written as

$$\phi(x) = \sum_{i=0}^{d-1} \phi_i \Lambda_i(x), \tag{1}$$

where [1]

$$d = \begin{cases} N & \text{if } \phi \text{ is periodic,} \\ N+p & \text{otherwise,} \end{cases}$$

and ϕ_i are the unknowns of the following matrix equation:

$$\sum_{i'=0}^{d-1} A_{ii'}\phi_{i'} = \rho_i, \qquad i = 0, \ldots, d-1. \tag{2}$$

Here the matrix $A_{ii'}$ and the right-hand-side ρ_i are respectively given by:

$$\begin{aligned} A_{ii'} &= \int_0^L dx\, C_1 \Lambda_i^{'} \Lambda_{i'}^{'} + \int_0^L dx\, C_2 \Lambda_i \Lambda_{i'}, \\ \rho_i &= \int_0^L dx \rho \Lambda_i. \end{aligned} \tag{3}$$

For more general differential operators, the matrix coefficients $A_{ii'}$ can be written as a sum of contributing matrices of the form

$$A_{ii'} = \int_0^L dx\, C \Lambda_i^{\alpha} \Lambda_{i'}^{\alpha'}, \tag{4}$$

where Λ_i^{α} denotes the α^{th} derivative of Λ_i. As the splines Λ_i have a support of $p+1$ intervals, the matrix is sparse and its coefficients $A_{ii'}$ are non-zero only for $|i - i'| \leq p$: hence the matrix has a band structure of bandwidth equal to $2p+1$ if the operator is purely differential. For an integral equation such as

$$\int_0^L dx' K(x, x')\phi(x') = \rho(x),$$

the discretization results in a *dense* matrix of the form

$$A_{ii'} = \int_0^L dx \Lambda_i(x) \int_0^L dx' K(x, x') \Lambda_{i'}(x'). \tag{5}$$

Note that when the kernel is separable $K(x, x') = U(x)V(x')$, the matrix $A_{ii'}$ is a *dyadic*:

$$A_{ii'} = \int_0^L dx U(x) \Lambda_i(x) \int_0^L dx V(x) \Lambda_{i'}(x) = U_i V_{i'}. \tag{6}$$

1.1.2 Periodic boundary conditions

The splines Λ_i are N-periodic ($\Lambda_{i+N}(x) = \Lambda_i(x-L)$). This property can be easily enforced while constructing both ρ_i and the matrix $A_{ii'}$. This results in a solution ϕ_i which is also N-periodic.

1.1.3 Non-periodic boundary conditions

In `BSPLINES` [1], the constructed non-periodic splines are such that at the boundaries $x = 0$ and $x = L$:

$$\Lambda_i(0) = \delta_{i,0}, \qquad \Lambda_i(L) = \delta_{i,N+p-1}, \tag{7}$$

which imply that, using (1)

$$\phi(0) = \phi_0, \qquad \phi(L) = \phi_{N+p-1}. \tag{8}$$

It is thus possible to impose the Dirichlet boundary conditions by a simple modification of the matrix $A_{ii'}$ as shown in Appendix B.1.

1.2 Problems in more dimensions

1.2.1 Two-dimensional equations

The results obtained above can be extended in a straightforward manner. Assuming, for example a *polar like* (r, θ) coordinate system, with the discretized solution and the right-hand side written as:

$$\begin{aligned} \phi(r,\theta) &= \sum_{i=0}^{N_r+p_r-1} \sum_{j=0}^{N_\theta-1} \phi_{ij} \Lambda_i(r) \Lambda_j(\theta) \\ \rho_{ij} &= \int_0^R dr \int_0^{2\pi} d\theta J(r,\theta) \rho(r,\theta) \Lambda_i(r) \Lambda_j(\theta), \end{aligned} \tag{9}$$

where $J(r, \theta)$ is the Jacobian, the matrix equation to solve is

$$\sum_{i'=0}^{N_r+p_r-1} \sum_{j'=0}^{N_\theta-1} A_{iji'j'} \phi_{i'j'} = \rho_{ij}, \tag{10}$$

with the matrix $A_{iji'j'}$ expressed as a sum of matrices of the form:

$$A_{iji'j'} = \int_0^R dr \int_0^{2\pi} d\theta \, C(r,\theta) \, \Lambda_i^\alpha(r) \Lambda_{i'}^{\alpha'}(r) \, \Lambda_j^\beta(\theta) \Lambda_{j'}^{\beta'}(\theta). \tag{11}$$

1.2.2 Three-dimensional equations

Likewise, for the three-dimension case, assuming for example a *toroidal like* (r, θ, φ) coordinate system, with the discretized solution and the right-hand side written as:

$$\begin{aligned} \phi(r,\theta,\varphi) &= \sum_{i=0}^{N_r+p_r-1} \sum_{j=0}^{N_\theta-1} \sum_{k=0}^{N_\varphi-1} \phi_{ijk} \Lambda_i(r) \Lambda_j(\theta) \Lambda_k(\varphi) \\ \rho_{ijk} &= \int_0^R dr \int_0^{2\pi} d\theta \int_0^{2\pi} d\varphi J(r,\theta,\varphi) \rho(r,\theta,\varphi) \Lambda_i(r) \Lambda_j(\theta) \Lambda_k(\varphi), \end{aligned} \tag{12}$$

where $J(r, \theta, \varphi)$ is the Jacobian, the matrix equation to solve is

$$\sum_{i'=0}^{N_r+p_r-1} \sum_{j'=0}^{N_\theta-1} \sum_{k'=0}^{N_\varphi-1} A_{ijki'j'k'} \phi_{i'j'k'} = \rho_{ijk}, \tag{13}$$

with the matrix $A_{ijki'j'k'}$ expressed as a sum of matrices of the form:

$$A_{ijki'j'k'} = \int_0^R dr \int_0^{2\pi} d\theta \int_0^{2\pi} d\varphi\, C(r,\theta,\varphi)\, \Lambda_i^{\alpha}(r)\Lambda_{i'}^{\alpha'}(r)\, \Lambda_j^{\beta}(\theta)\Lambda_{j'}^{\beta'}(\theta)\Lambda_k^{\gamma}(\varphi)\Lambda_{k'}^{\gamma'}(\varphi). \tag{14}$$

1.2.3 Unicity condition

In the case of the polar coordinates (r,θ) considered above, the unicity condition on the axis $r = 0$ should be imposed. It can be enforced by modifications of the matrix A as described in Appendix B.2.

1.2.4 One-dimensional numbering

For two-dimensional (three-dimensional) problems, the solution ϕ_{ij} (ϕ_{ijk}) as well as the right-hand-side ρ_{ij} (ρ_{ijk}) can be conveniently casted into one-dimensional arrays. As an example, by numbering first the last index, we obtain the following mappings:

$$\mu = \begin{cases} j + iN_\theta & \text{two-dimensional case} \\ k + (j + iN_\theta)N_\varphi & \text{three-dimensional case} \end{cases} \tag{15}$$

Using such a one-dimensional numbering, the matrix equation for the two and three dimensional cases takes a more conventional form:

$$\sum_{\mu'=0}^{r-1} A_{\mu\mu'}\phi_{\mu'} = \rho_\mu, \tag{16}$$

with the respective matrix ranks $r = (N_r + p_r)N_\theta$ and $r = (N_r + p_r)N_\theta N_\varphi$. For a pure differential operator, the matrix $A_{\mu\mu'}$ has a band structure of bandwidth $b = 2(p_r + 1)N_\theta - 1$ and $b = 2(p_r + 1)N_\theta N_\varphi - 1$ respectively. It is important to note that, except for the one-dimensional problem, there are *many* zeros inside the matrix band!

2 The module MATRIX

2.1 Interface

The Fortran module `MATRIX` contains easy-to-use routines to solve the matrix equation formulated in the previous section, using the direct solvers of LAPACK. The different matrix storage formats are defined, using the Fortran derived datatypes. The different types in the present version are listed in Table 1.

`gemat`	General dense matrix
`gbmat`	General band matrix
`pbmat`	Symmetric positive-definite band matrix
`periodic_mat`	Matrix obtained for example in one-dimensional periodic problems

Table 1: The matrix types

These types define `DOUBLE PRECISION` real matrices. `DOUBLE COMPLEX` matrices are declared by prefixing these types with the letter "`z`", for example `zgbmat`. Note that `zpbmat` defines a *hermitian* positive-definite complex matrix. The *generic* routines are defined for each of these types in Table 2. Note that the routine `updtmat` is mainly used for the matrix assembly while `getxxx` and `putxxx` are rather used to modify the matrix, for example to impose boundary conditions.

The complete description of each routine is given in Appendix C. More information on how to use it can be obtained by `greping` its name on the examples found in the `examples/` directory.

`init`	Initializes the data structure
`destroy`	Free the data structure memory
`updtmat`	Accumulates a value to the element A_{ij}
`putele, putrow, putcol`	Overwrites the matrix element (i,j), row i, column j
`getele, getrow, getcol`	Returns the matrix element (i,j), row i, column j
`vmx`	Returns the matrix-vector product
`mcopy`	Copy a matrix to another matrix
`maddto`	Adds 2 matrices: $A \leftarrow A + \alpha B$
`determinant`	Returns the matrix determinant
`factor`	Computes the LU (Cholesky for symmetric/hermitian matrix) factorization
`bsolve`	Solves the linear system using the factorized matrix

Table 2: The generic routines in the `MATRIX` module

2.2 A two-dimensional example

Let's consider the Poisson equation using the cylindrical coordinates (r,θ):

$$-\frac{1}{r}\frac{\partial}{\partial r}\left[r\frac{\partial \phi}{\partial r}\right] - \frac{1}{r^2}\frac{\partial^2 \phi}{\partial \theta^2} = \rho, \qquad \phi(r=1,\theta) = 0. \tag{17}$$

Assuming the exact solution

$$\phi(r,\theta) = (1-r^2)r^m \cos m\theta,$$

the right-hand-side becomes

$$\rho = 4(m+1)r^m \cos m\theta.$$

The matrix and the right hand-side of the discretized problem are computed as

$$\begin{aligned} A_{iji'j'} &= \int_0^1 dr \int_0^{2\pi} d\theta \left[r\,\Lambda_i'(r)\Lambda_{i'}'(r)\,\Lambda_j(\theta)\Lambda_{j'}(\theta) + \frac{1}{r}\,\Lambda_i(r)\Lambda_{i'}(r)\,\Lambda_j'(\theta)\Lambda_{j'}'(\theta)\right] \\ \rho_{ij} &= \int_0^1 dr \int_0^{2\pi} d\theta\; r\rho(r,\theta)\;\Lambda_i(r)\Lambda_j(\theta). \end{aligned} \tag{18}$$

In the example `pde2d.f90` this problem is treated in detail. In the following, only the calls to the `MATRIX` routines are reviewed to show how the matrix problem is solved using the `MATRIX` module.

```
!
!  Declare a General Band matrix
!
  USE matrix
  USE conmat_mod
  TYPE(gbmat) :: mat
!
!  Rank and bandwidth. nidbas(1) is the spline order in
!  the first dimension r.
!
  nrank = (nx+nidbas(1))*ny    ! Rank of the FE matrix
  kl = (nidbas(1)+1)*ny -1     ! Number of sub-diagnonals
  ku = kl                      ! Number of super-diagnonals
  nterms = 2                   ! Number of terms in the weak form
```

```
!
!  Initialize the matrix data structure
!
  CALL init(kl, ku, nrank, nterms, mat)
!
!  Construct the matrix, using 2D spline splxy
!  and impose boundary conditions
!
  CALL conmat(splxy, mat, coefeq_poisson)
  CALL ibcmat(mat, ny)
!
!  Compute the RHS, using the 2D spline splxy
!  and impose boundary conditions
!
  CALL disrhs(mbess, splxy, rhs)
  CALL ibcrhs(rhs, ny)
!
!   Factor the matrix and solve
!
  CALL factor(mat)
  CALL bsolve(mat, rhs, sol)
!
...
CONTAINS
  SUBROUTINE coefeq_poisson(x, y, idt, idw, c)
    DOUBLE PRECISION, INTENT(in) :: x, y
    INTEGER, INTENT(out) :: idt(:,:), idw(:,:)
    DOUBLE PRECISION, INTENT(out) :: c(:)
!
    c(1) = x
    idt(1,1) = 1; idt(1,2) = 0
    idw(1,1) = 1; idw(1,2) = 0
!
    c(2) = 1.d0/x
    idt(2,1) = 0;  idt(2,2) = 1
    idw(2,1) = 0;  idw(2,2) = 1
  END SUBROUTINE coefeq_poisson
```

Some explanations and remarks:

- The matrix construction is performed by `conmat` which will be described later in section 6. The *weak form* is defined in the *internal* procedure `coefeq_poisson` and passed as an argument to `conmat`. See section 6 for a detailed description of the variables `c, idt, idw` returned by `coefeq_poisson`.
- Boundary conditions are imposed by modifications of the matrix in subroutine `ibcmat` (see file `the ibcmat.f90`), using the `MATRIX` routines `getrow, putrow, getcol, putcol`.
- The construction of the right-hand-side in `disrhs` (see the file `disrhs.f90`) is computed using a Gauss quadrature..
- Using the `pbmat` type instead of `gbmat` (the matrix in this example is symmetric and positive-definite!) requires only a few modifications of the program (see the complete example `pde2d_pb.f90`):

  - Change the type `gbmat` to `pbmat` in all matrix declarations
  - Change the list of arguments in the routine `init` to (`ku, nrank, nterms, mat`)
  - Small changes in the boundary conditions (`ibcmat` and `ibcrhs`) to take into account the symmetry.

- The module `MATRIX` can be used independently of `BSPLINES` (which is used here only to compute the matrix and right-hand-side), for example in a problem discretized using Finite Differences.

3 Sparse matrix storage

Using the *band matrix format* for a pure differential operator requires to store a full bandwidth $b = 2(p_r + 1)N_\theta - 1$ for the two-dimensional problem as shown in section 1, while there are only $(2p_r + 1)^2$ non-zero elements per matrix row. In three-dimensional problem, it is much worse since $b = 2(p_r + 1)N_\theta N\varphi - 1$ for $(2p_r + 1)^3$ non-zero elements.

In order to reduce the matrix storage, a solution consists of just storing the matrix non-zero elements and use the *Sparse Direct Solvers*. With an optimal *renumbering* strategy (or *fill-in reducing ordering*), the size of the factored matrix can be expected to be smaller than the corresponding band matrix.

An alternative is to use *iterative* methods which usually need less memory.

Such a sparse matrix is implemented in the `SPARSE` module where each matrix row is represented by a `linked` list of elements with sorted column index. The data structure of this sparse matrix is wrapped up in a the Fortran data type `spmat` for a real matrix and `zspmat` for a complex matrix. Most of the generic routines which are already defined in the `MATRIX` module are overloaded for these matrix types. They are listed in Table 3. The complete documentation of these routines can be found in Appendix D.

Matrix types	`spmat, zspmat`
`init`	Initializes the data structure
`destroy`	Free the data structure memory
`updtmat`	Accumulates a value to the element A_{ij}
`putele, putrow, putcol`	Overwrites the matrix element (i, j), row i, column j
`getele, getrow, getcol`	Returns the matrix element (i, j), row i, column j
`get_count`	Get the number of non-zero elements in matrix

Table 3: The generic routines in the `SPARSE` module

It should be noted that this module is *not* used directly in solver problems. One usually uses instead modules which are specific to a type of (direct or iterative) solver. As will be shown in the next section, it is the routines in this solver module which directly calls the routines defined in the `SPARSE` module during the matrix assembly.

4 Solvers using the module `SPARSE`

All the solvers discussed in this section use initially the module `SPARSE` to construct the sparse matrix. Once this construction procedure is complete, this matrix is converted to the (usually more efficient) format used by the solver. In a time-dependent simulation where the problem matrix changes but not the sparsity pattern, the subsequent matrix assembly will be performed directly on this solver's format.

Thus for example, the first time `updtmat` is called on a new matrix, it is the version from `SPARSE`. Next, if `updtmat` is called again to modify the matrix, it will be the solver's version, unless the matrix is re-initialized by a call to `destroy` followed by `init`. This switch is completely transparent for the user as shown through an example in the next section.

4.1 The PARDISO direct solver

The interface to PARDISO [2] is implemented in the module PARDISO_BSPLINES. The matrix type (symmetric, hermitian, positive-definite) is set by the arguments nlsym, nlherm and nlpos passed to the generic routine init. All the other generic routines defined in the module MATRIX, plus routines specific to the sparse solver, are listed in Table 4. The complete documentation of these routines is given in Appendix E.

Matrix types	pardiso_mat, zpardiso_mat
init	Initializes the data structure
destroy	Free the data structure memory
updtmat	Accumulates a value to the element A_{ij}
putele, putrow, putcol	Overwrites the matrix element (i, j), row i, column j
getele, getrow, getcol	Returns the matrix element (i, j), row i, column j
vmx	Returns the matrix-vector product
mcopy	Copy a matrix to another matrix
maddto	Adds 2 matrices: $A \leftarrow A + \alpha B$
clear_mat	Set the matrix elements to zero.
psum_mat	Parallel sum of matrices
p2p_mat	Point-to-point combine sparse matrix between 2 processes
get_count	Get the number of non-zero elements in matrix
factor	Factorization
bsolve	Solves the linear system using the factorized matrix
to_mat	Convert to PARDISO CSR matrix format
reord_mat	Reordering and symbolic factorization
numfact	Numerical factorization

Table 4: The generic routines in the PARDISO_BSPLINES module

Below, a complete example solving a simple two-dimensional Poisson discretized by the 5 point Finite Difference method illustrates how to use the PARDISO_BSPLINES module.

```
PROGRAM main
  USE pardiso_bsplines
  IMPLICIT NONE
  TYPE(pardiso_mat) :: amat
  DOUBLE PRECISION, ALLOCATABLE :: rhs(:), sol(:), arow(:)
  INTEGER :: nx=5, ny=4
  INTEGER :: n, nnz
  INTEGER :: i, j, irow, jcol
!
  WRITE(*,'(a)', advance='no') 'Enter nx, ny: '
  READ(*,*) nx, ny
  n = nx*ny  ! Rank of the matrix
  ALLOCATE(rhs(n))
  ALLOCATE(sol(n))
  ALLOCATE(arow(n))
!
  CALL init(n, 1, amat, nlsym=.TRUE.)  ! Pardiso mat, symmetric case
!
!  Construct the matrix and RHS
!
```

```
  DO j=1,ny
     DO i=1,nx
        arow = 0.0d0
        irow = numb(i,j)
        arow(irow) = 4.0d0
        IF(i.GT.1)  arow(numb(i-1,j)) = -1.0d0
        IF(i.LT.nx) arow(numb(i+1,j)) = -1.0d0
        IF(j.GT.1)  arow(numb(i,j-1)) = -1.0d0
        IF(j.LT.ny) arow(numb(i,j+1)) = -1.0d0
        CALL putrow(amat, irow, arow)
        rhs(irow) = SUM(arow)   ! => the exact solution is 1
     END DO
  END DO
!
  WRITE(*,'(a,i6)') 'Number of non-zeros of matrix', get_count(amat)
!
!  Factor the amat matrix (Reordering, symbolic and numerical factorization)
!
  CALL factor(amat)
!
!  Back solve
!
  CALL bsolve(amat, rhs, sol)
!
!   Check solutions
!
  WRITE(*,'(/a/(10f8.4))') 'Computed sol', sol
  WRITE(*,'(a,1pe12.3)') 'Error', MAXVAL(ABS(sol-1.0d0))
!
!   Clean up
!
  DEALLOCATE(rhs)
  DEALLOCATE(sol)
  DEALLOCATE(arow)
  CALL destroy(amat)
CONTAINS
  INTEGER FUNCTION numb(i,j)
!
!  One-dimensional numbering
!  Number first x then y
!
    INTEGER, INTENT(in) :: i, j
    numb = (j-1)*nx + i
  END FUNCTION numb
END PROGRAM main
```

It should be noted that

- The routine `putrow` in the matrix construction loop uses the version from the `SPARSE` module to create dynamically the matrix row using the linked list.

- The routine `factor` calls successively the matrix conversion `to_mat`, the reordering and symbolic factorization routine `reord_mat` and finally the numerical factorization `numfact`. One could indeed call these three routines separately instead of the single call to `factor`,
- After solving the linear system, if the matrix is modified by calling for example `putrow` again, it will modify directly the converted matrix and not on the `spmat` matrix which is anyway *destroyed* at the end of `to_mat`.
- If the matrix sparsity changes, the matrix should be re-initialized by calling the `destroy` and `init` routines.

Other examples can be found by running "`grep pardiso_mat`" on the F90 files in the directory `examples/`.

4.2 The WSMP direct solver

The interface to WSMP [3] is implemented in the module `WSMP_BSPLINES`. The matrix type (symmetric, hermitian, positive-definite) is set by the arguments `nlsym, nlherm` and `nlpos` passed to the generic routine `init`. All the other generic routines defined in the module MATRIX, plus routines specific to the sparse solver, are listed in Table 5. The complete documentation of these routines is given in Appendix F.

Matrix types	`wsmp_mat, zwsmp_mat`
`init`	Initializes the data structure
`destroy`	Free the data structure memory
`updtmat`	Accumulates a value to the element A_{ij}
`putele, putrow, putcol`	Overwrites the matrix element (i, j), row i, column j
`getele, getrow, getcol`	Returns the matrix element (i, j), row i, column j
`vmx`	Returns the matrix-vector product
`mcopy`	Copy a matrix to another matrix
`maddto`	Adds 2 matrices: $A \leftarrow A + \alpha B$
`clear_mat`	Set the matrix elements to zero.
`psum_mat`	Parallel sum of matrices
`p2p_mat`	Point-to-point combine sparse matrix between 2 processes
`get_count`	Get the number of non-zero elements in matrix
`factor`	Factorization
`bsolve`	Solves the linear system using the factorized matrix
`to_mat`	Convert to WSMP CSR matrix format
`reord_mat`	Reordering and symbolic factorization
`numfact`	Numerical factorization

Table 5: The generic routines in the `WSMP_BSPLINES` module

The simple Poisson example using the `PARDISO_BSPLINES` module shown in the previous section can be easily adapted to the WSMP interface since there are only two lines to change: the `USE` and the matrix `TYPE` lines.

Other examples of how to use this interface can be found by running "`grep wsmp_mat`" on the F90 files the directory `examples/`.

The same solver functionality can be found in both the PARDISO and WSMP solvers as one can verify by comparing Table 4 and Table 5 or the description of routines in Appendix E and Appendix F. However, there is an important difference. While in PARDISO (and indeed also in LAPACK), it is possible to define several matrices to solve simultaneously, it appears that in WSMP, this is possible *only* for symmetric and hermitian matrices: in the present 10.9 version, the routines to store and recall the solver context

WSTOREMAT/WRECALLMAT which are present in the symmetric version of the library are missing in the general version!

A separate module named PWSMP_BSPLINES added the MPI *parallelization* capability provided by WSMP. This parallel version implements the same user interface as shown in Table 5. The following considerations should be however taken in to account:

1. The coefficient matrix amat is partitioned into blocks of *contiguous* rows, with their indices defined in the interval [amat%istart,amat%iend] which is defined after the call to init.
2. Calls to the routine updtmat to update the matrix coefficients should not specify a row index *outside* this interval. On the other hand, getxxx will return 0 and putxxx will ignore it if a row index *not* in the range [amat%istart,amat%iend] is passed to them.
3. An *optional* MPI communicator can be given to init using the keyword comm_in. By default, the communicator MPI_COMM_WORLD is used.

A complete example using PWSMP_BSPLINES can be found in examples/pde2d_pwsmp.f90.

4.3 The MUMPS direct solver

MUMPS [4] is a *parallel sparse direct solver* using MPI and is implemented in the module MUMPS_BSPLINES. User program using this module should be compiled and linked with MPI even if only the serial version of the solver is needed, in which case the MPI_COMM_SELF is passed to the initialization routine init as an *optional* argument with the keyword comm_in. Otherwise a valid MPI communicator should be passed. By default comm_in=MPI_COMM_SELF. Note that it is possible to use both serial and parallel solvers in the same program to solve different matrix problems.

As for PARDISO and WSMP, the same user interface to the MUMPS solver is used and summarized in Table 6. The complete documentation of these routines is given in Appendix G.

Matrix types	mumps_mat, zmumps_mat
init	Initializes the data structure
destroy	Free the data structure memory
updtmat	Accumulates a value to the element A_{ij}
putele, putrow, putcol	Overwrites the matrix element (i, j), row i, column j
getele, getrow, getcol	Returns the matrix element (i, j), row i, column j
vmx	Returns the matrix-vector product
mcopy	Copy a matrix to another matrix
maddto	Adds 2 matrices: $A \leftarrow A + \alpha B$
clear_mat	Set the matrix elements to zero.
psum_mat	Parallel sum of matrices
p2p_mat	Point-to-point combine sparse matrix between 2 processes
get_count	Get the number of non-zero elements in matrix
factor	Factorization
bsolve	Solves the linear system using the factorized matrix
to_mat	Convert to WSMP CSR matrix format
reord_mat	Reordering and symbolic factorization
numfact	Numerical factorization

Table 6: The generic routines in the MUMPS_BSPLINES module

5 Fourier solver [5]

5.1 The matrix equation in Fourier space

For a periodic one-dimensional problem, the solution ϕ_i and the right-hand-side ρ_i in (2) are N-periodic. Their Discrete Fourier Transform (DFT) can be defined as

$$\begin{aligned} \hat{\phi}_k &= \sum_{j=0}^{N-1} \phi_j e^{i\frac{2\pi}{N}kj}, \qquad \hat{\rho}_k = \sum_{j=0}^{N-1} \rho_j e^{i\frac{2\pi}{N}kj}, \\ \phi_j &= \frac{1}{N}\sum_{k=0}^{N-1} \hat{\phi}_k e^{-i\frac{2\pi}{N}kj}, \qquad \rho_j = \frac{1}{N}\sum_{k=0}^{N-1} \hat{\rho}_k e^{-i\frac{2\pi}{N}kj}. \end{aligned} \tag{19}$$

Taking the DFT of Eq. (2), we obtain the following matrix equation in Fourier space:

$$\sum_{k'=0}^{N-1} \hat{A}_{kk'}\hat{\phi}_{k'} = \hat{\rho}_k, \tag{20}$$

where $\hat{A}_{kk'}$ is the DFT of the original matrix. Following the notations in Eq. (4) and assuming an *equidistant* mesh with the interval $\Delta = L/N$, each of the DFT matrices of the weak form can be written as

$$\begin{aligned} \hat{A}_{kk'} &= \frac{1}{N}\sum_{j=0}^{N-1} e^{i\frac{2\pi}{N}kj} \sum_{j'=0}^{N-1} A_{jj'} e^{-i\frac{2\pi}{N}k'j'} \\ &= \frac{1}{N}\int_0^L dx\, C(x) \sum_{j=0}^{N-1} e^{i\frac{2\pi}{N}kj}\Lambda_j^\alpha(x) \sum_{j'=0}^{N-1} e^{-i\frac{2\pi}{N}k'j'}\Lambda_{j'}^{\alpha'}(x) \\ &= \frac{1}{N}\sum_{J=0}^{N-1}\int_{J\Delta}^{(J+1)\Delta} dx\, C(x) \sum_{j=0}^{N-1} e^{i\frac{2\pi}{N}kj}\Lambda_j^\alpha(x) \sum_{j'=0}^{N-1} e^{-i\frac{2\pi}{N}k'j'}\Lambda_{j'}^{\alpha'}(x) \end{aligned} \tag{21}$$

Note that each of the last two sums is over the splines which are non-zero at a given x. Using the translational symmetry of the periodic splines:

$$\sum_j e^{i\frac{2\pi}{N}kj}\Lambda_j^\alpha(x) = \sum_j e^{i\frac{2\pi}{N}kj}\Lambda_{j-J}^\alpha(x - J\Delta) = e^{i\frac{2\pi}{N}kJ}\,\hat{\Lambda}_k^\alpha(x - J\Delta),$$

where we have defined the DFT of splines $\hat{\Lambda}_k(x)$ as

$$\hat{\Lambda}_k^\alpha(x) = \sum_j \Lambda_j^\alpha(x) e^{i\frac{2\pi}{N}kj}, \tag{22}$$

which are computed by the routine `ft_basfun` in the module `BSPLINES` for any spline order p and derivative $\alpha \le p$. The DFT matrices can now be written as:

$$\hat{A}_{kk'} = \frac{1}{N}\sum_{J=0}^{N-1}\int_{J\Delta}^{(J+1)\Delta} dx\, C(x)\, e^{i\frac{2\pi}{N}J(k-k')}\hat{\Lambda}_k^\alpha(x - J\Delta)\,\left[\hat{\Lambda}_{k'}^{\alpha'}(x - J\Delta)\right]^*.$$

Finally, making the variable transform $x \to x + J\Delta$ and defining the DFT of the weak-form coefficient C as

$$\hat{C}_k(x) = \sum_{J=0}^{N-1} C(x + J\Delta)\, e^{i\frac{2\pi}{N}Jk}, \tag{23}$$

the DFT of the matrix $\hat{A}_{kk'}$ can be calculated as an integration over the first interval:

$$\hat{A}_{kk'} = \frac{1}{N}\int_0^\Delta dx\, \hat{C}_{k-k'}(x)\,\hat{\Lambda}_k^\alpha(x)\left[\hat{\Lambda}_{k'}^{\alpha'}(x)\right]^*, \tag{24}$$

which can be computed using again the same Gauss formula as before. For uniform C, $\hat{A}_{kk'}$ is diagonal and the matrix equation (20) reduces to a system of equations for the uncoupled Fourier modes.

When C is non-uniform, $\hat{A}_{kk'}$ is *dense*. However in problems where the solution is expected to be "smooth", one can keep only a small number of Fourier modes, reducing thus the rank of $\hat{A}_{kk'}$. Furthermore, if the coefficients $C(x)$ of the differential equations are very smooth, peaked at a few (low order) modes, the DFT matrix can become *sparse*!

The generalization to the two-dimensional problem (11) is straightforward:

$$\hat{A}_{im,i'm'} = \frac{1}{N_\theta}\int_0^R dr \left\{ \int_0^{\Delta\theta} d\theta\, \hat{C}_{m-m'}(r,\theta)\, \hat{\Lambda}^{\beta}_{m}(\theta) \left[\hat{\Lambda}^{\beta'}_{m'}(\theta)\right]^* \right\} \Lambda^{\alpha}_{i}(r)\Lambda^{\alpha'}_{i'}(r) \tag{25}$$

$$\hat{C}_m(r,\theta) = \sum_{j=0}^{N_\theta-1} C(r,\theta+j\Delta\theta)\, e^{i\frac{2\pi}{N_\theta}jm}. \tag{26}$$

Likewise, for the three-dimensional problem (14), we obtain

$$\hat{A}_{imn,i'm'n'} = \frac{1}{N_\theta N_\varphi}\int_0^R dr \left\{ \int_0^{\Delta\theta} d\theta \int_0^{\Delta\varphi} d\varphi\, \hat{C}_{m-m',n-n'}(r,\theta,\varphi)\, \hat{\Lambda}^{\beta}_{m}(\theta) \left[\hat{\Lambda}^{\beta'}_{m'}(\theta)\right]^* \hat{\Lambda}^{\gamma}_{n}(\varphi) \left[\hat{\Lambda}^{\gamma'}_{n'}(\varphi)\right]^* \right\} \Lambda^{\alpha}_{i}(r)\Lambda^{\alpha}_{i'}(r) \tag{27}$$

$$\hat{C}_{mn}(r,\theta,\varphi) = \sum_{j=0}^{N_\theta-1}\sum_{k=0}^{N_\varphi-1} C(r,\theta+j\Delta\theta,\varphi+k\Delta\varphi)\, e^{i\frac{2\pi}{N_\theta}jm}\, e^{i\frac{2\pi}{N_\varphi}kn}. \tag{28}$$

Note that for axi-symmetric systems where the coefficients C do not depend on φ

$$\hat{C}_{mn} = \hat{C}_{mn}\delta_{n,0} \tag{29}$$

and thus the three-dimensional problem reduces to a set of independent two-dimensional problems with

$$\begin{aligned} \hat{A}^{n}_{im.i'm'} &= M_n \hat{A}_{im.i'm'} \\ M_n &= \int_0^{\Delta\varphi} d\varphi \left|\hat{\Lambda}_n(\varphi)\right|^2. \end{aligned} \tag{30}$$

5.2 Integral equation

The DFT matrices for differential operators derived above can be extended to an integral operator of the following form:

$$\int_0^L dx' K(x,x')\,\phi(x') = \rho(x), \tag{31}$$

where $\phi(x)$ is L-periodic. Using the same FE discretization as above results in the following matrix in *real* space:

$$A_{jj'} = \int_0^L dx\, \Lambda_j(x) \int_0^L dx'\, K(x,x')\, \Lambda_{j'}(x'), \tag{32}$$

and its DFT

$$\hat{A}_{kk'} = \frac{1}{N}\sum_{J=0}^{N-1}\int_{J\Delta}^{(J+1)\Delta} dx \sum_{J'=0}^{N-1}\int_{J'\Delta}^{(J'+1)\Delta} dx'\, K(x,x')\, e^{i\frac{2\pi}{N}kJ}\hat{\Lambda}_k(x-J\Delta)\, e^{-i\frac{2\pi}{N}k'J'} \left[\hat{\Lambda}_{k'}(x-J'\Delta)\right]^*, \tag{33}$$

Now, defining the DFT of the kernel as

$$\hat{K}_{kk'}(x,x') = \sum_{J=0}^{N-1}\sum_{J'=0}^{N-1} K(x+J\Delta,x'+J'\Delta)\, e^{i\frac{2\pi}{N}kJ}\, e^{-i\frac{2\pi}{N}k'J'}, \tag{34}$$

the final expression for the DFT of the matrix $\hat{A}_{kk'}$ reduces to

$$\hat{A}_{kk'} = \frac{1}{N}\int_0^{\Delta} dx \int_0^{\Delta} dx'\, \hat{K}_{kk'}(x,x')\,\hat{\Lambda}_k(x)\,\left[\hat{\Lambda}_{k'}(x')\right]^*. \tag{35}$$

Again, notice that the dense matrix $\hat{A}$ can become *sparse* if only a limited number of Fourier modes are retained in the DFT of the kernel $\hat{K}$.

5.3 A two-dimensional example with a non-uniform coefficient

As a check, we considered here a two-dimensional example similar to the example in section 2.2 but with a non-uniform coefficient:

$$-\frac{1}{r}\frac{\partial}{\partial r}\left[rC\frac{\partial \phi}{\partial r}\right] - \frac{1}{r^2}\frac{\partial}{\partial \theta}\left[C\frac{\partial \phi}{\partial \theta}\right] = \rho. \tag{36}$$

With $C(r,\theta) = 1 + \epsilon r\cos\theta$, assuming the same exact solution as in section (2.2)

$$\phi(r,\theta) = (1-r^2)r^m\cos m\theta, \tag{37}$$

the right-hand side becomes

$$\begin{aligned}\rho(r,\theta) = 4(m+1)r^m\cos m\theta + \frac{\epsilon r^m}{2}(4+5m-m/r^2)\cos(m-1)\theta \\ + \frac{\epsilon r^m}{2}(4+3m+m/r^2)\cos(m+1)\theta.\end{aligned} \tag{38}$$

This problem is solved in real space and Fourier space respectively in example `pde2d_sym_pardiso.f90` and example `pde2d_sym_pardiso_dft.f90`. Both use the `PARDISO_BSPLINES` module to solve the sparse matrix equation. It should be noted that the Fourier method should yield the *same solution* as found with the solver in real space if *all* the N_θ Fourier modes are kept.

For the problem defined above with `m=3`, by keeping only the seven Fourier modes in $[-3,3]$ and the three mode couplings $[-1,0,1]$ in the Fourier solver, we found that both methods yield the same (up to 5 digits) *relative discretization error*. Furthermore, increasing the number of Fourier modes to $[-4,4]$ (note that the $m = \pm 4$ Fourier components of the right hand side ρ are not null) does not increase the accuracy of the computed solution!

In this example, the matrix in Fourier space has a rank which is $N_\theta/7$ times smaller than in the solver in real space. The number of non-zeros is also reduced by a factor of $(2p+1)/3$ since only 3 Fourier mode coupling terms are considered.

In general, the efficiency of such a *matrix filter* is expected to be problem-dependent. The Fourier solver should be tested in real simulations.

6 The matrix construction module CONMAT_MOD

The module implements the generic matrix construction subroutine `conmat`, using the algorithm detailed in Appendix A, for 1D and 2D differential equations. The computed matrix is returned in the argument `mat` which can be a Lapack band matrix as well as a PARDISO, WSMP or MUMPS sparse matrix. The complete interface of the subroutine is given below.

```
SUBROUTINE conmat(spl, mat, coefeq, maxder)
  TYPE(spline1d|spline2d) :: spl
  TYPE([z]gbmat|[z]pbmat|[z]periodic_mat|[z]pardiso|...) :: mat
  INTEGER, INTENT(in), OPTIONAL :: maxder[(2)]
  INTERFACE
   SUBROUTINE coefeq(x, [y], idt, idw, c)
     DOUBLE PRECISION, INTENT(in) :: x, [y]
     INTEGER, INTENT(out) :: idt(:), idw(:)
     DOUBLE PRECISION, INTENT(out) :: c(:)
   END SUBROUTINE coefeq
  END INTERFACE
```

Purpose: Construct the FE matrix for 1D or 2D differential operator.

Arguments:

```
spl    : 1D or 2D spline
mat    : matrix object
coefeq : user provided subroutine (see below)
maxder : Maximum order of the derivatives in the weak form.
         Equal to 1 (first derivative) by default.
```

The subroutine `conmat` includes, in addition to the arguments `spl, mat` and `maxder` described above, an user provided subroutine as the third argument `coefeq` which computes all the weak form coefficients defined in Eq.(39) and Eq.(42) for a given point (x for 1D case or (x, y) for 2D case). The output array `c` will contain all the computed C with its corresponding derivative orders (d, d') returned in `idt, idw` respectively. Other quantities required to calculate the coefficients C could be communicated to `coefeq`, using for example a `COMMON` block or a `MODULE`.

An example of using this module can be found in section 2.2.

A Matrix assembly for differential operators

A.1 1D case

A.1.1 Local matrix

The contribution to the discretized weak-form from the interval $[x_i, x_{i+1}]$ where $i = 1, \ldots, N$, is a sum of the *local matrices*

$$\begin{aligned} A^i_{\alpha\alpha'} &= \int_{x_i}^{x_{i+1}} dx\, C(x)\Lambda^d_\alpha(x)\Lambda^{d'}_{\alpha'}(x) \\ &\simeq \sum_{g=1}^{G} \underbrace{w_g \Lambda^d_\alpha(x_g)\Lambda^{d'}_{\alpha'}(x_g)}_{F_{\alpha\alpha' g}} \underbrace{C(x_g)}_{c_g}, \end{aligned} \tag{39}$$

where a G point Gauss quadrature over the interval $[x_i, x_{i+1}]$ is used to approximate the integral and Λ^d_α denotes the d^{th} derivative of splines which are non zero in the interval $[x_i, x_{i+1}]$. For splines of degree p, $\alpha = 0, \ldots, p$. Note that the matrix can be written as a *matrix-vector product*:

$$\mathbf{A} = \mathbf{F} \cdot \mathbf{c}. \tag{40}$$

A.1.2 Mapping to global matrix

For N intervals, the number of spline elements of degree p, is $N_e = N + p$, or $N_e = N$ if the system is *periodic*. Once the local matrix $A_{\alpha\alpha'}$ is formed, it can be added to the *global* matrix using the mapping:

$$\begin{aligned} A^g_{II'} &\leftarrow A^g_{II'} + A^i_{\alpha\alpha'} \\ I = i + \alpha, &\qquad I' = i + \alpha' \end{aligned} \tag{41}$$

For periodic problems, the indices I, I' are further transformed by taking into account the periodicity N, using for example the following `FORTRAN` statement

```
I = MODULO(I-1,N) + 1
```

A.2 2D case

A.2.1 Local matrix

In this case, the local matrix obtained for the grid cell $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$ takes the form:

$$\begin{aligned} A_{\alpha\alpha'\beta\beta'} &= \int_{x_i}^{x_{i+1}} dx \int_{y_j}^{y_{j+1}} dy\, \Lambda^{d_1}_{\alpha}(x)\Lambda^{d_1'}_{\alpha'}(x)\; C(x,y)\; \Lambda^{d_2}_{\beta}(y)\Lambda^{d_2'}_{\beta'}(y) \\ &\simeq \sum_{g_1=1}^{G_1} \underbrace{w_{g_1}\Lambda^{d_1}_{\alpha}(x_{g_1})\Lambda^{d_1{}'}_{\alpha'}(x_{g_1})}_{F_{\alpha\alpha' g_1}} \sum_{g_2=1}^{G_2} \underbrace{C(x_{g_1}, y_{g_2})}_{C_{g_1 g_2}} \underbrace{w_{g_2}\Lambda^{d_2}_{\beta}(y_{g_2})\Lambda^{d_2{}'}_{\beta'}(y_{g_2})}_{G_{\beta\beta' g_2}}, \end{aligned} \tag{42}$$

which can be computed efficiently as *matrix-matrix products*

$$\mathbf{A} = \mathbf{F} \cdot \mathbf{C} \cdot \mathbf{G}^{\mathbf{T}} \tag{43}$$

A.2.2 Mapping to global matrix

The local to global element indices mapping on each of the two dimensions can be defined as previously as

$$\begin{aligned} I = i + \alpha, &\qquad I' = i + \alpha' \\ J = j + \beta, &\qquad J' = j + \beta' \end{aligned} \tag{44}$$

If any of the 2 dimensions is periodic, the periodic condition have to be applied to the corresponding global element index as explained above.

Furthermore, in order to reduce the 4 dimension array $A^g_{II'JJ'}$ to the standard 2 dimension matrix, we number first the elements in y coordinate and obtain the following index transformation:

$$\mu = J + N^y_e(I-1), \qquad \mu' = J' + N^y_e(I'-1), \tag{45}$$

where N^y_e is the number of elements along the y coordinate. The *global* matrix is then constructed from

$$A^g_{\mu\mu'} \leftarrow A^g_{\mu\mu'} + A_{\alpha\alpha'\beta\beta'} \tag{46}$$

B The boundary conditions

B.1 Dirichlet condition

B.1.1 1D case

Let us consider the boundary condition $u(0) = c$. Since all the splines are 0 at $x = 0$, except for the first spline which is equal to 1, $\Lambda_i(0) = \delta_{i,1}$, we have simply

$$c = u(0) = \sum_{i=1}^{N} u_i \Lambda_i(0) \quad \Longrightarrow \quad u_1 = c. \tag{47}$$

The discretized linear system of equations, taking into account of this BC, can thus be written as

$$\begin{aligned} u_1 &= c \\ \sum_{j=2}^{N} A_{ij} u_j &= f_i - A_{i1} c, \quad i = 2, \ldots, N \end{aligned} \tag{48}$$

or in the following matrix form:

$$\begin{pmatrix} 1 & 0 & \cdots \\ 0 & A_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} c \\ f_2 - cA_{21} \\ \vdots \\ f_N - cA_{N1} \end{pmatrix} \tag{49}$$

Note that (1) the transformed matrix preserves any symmetry or positivity of the original matrix, (2) the first column of the original matrix has to be saved in order to modify the RHS f_i but only for non zero c and (3) in that case, one needs to save only $[A_{i1}]_{i=2}^{p+1}$, where p is the spline order.

In summary, the procedure for imposing the Dirichlet BC $u_1 = c$ can be summarized as follows:

1. Matrix transformation
   (a) Clear the matrix row $i = 1$ and set its diagonal term A_{11} to 1.
   (b) Get the matrix column $A_{j1}, \quad j = 2, \ldots, p + 1$ and save it.
   (c) Clear the matrix column $j = 1$ and set its diagonal term A_{11} to 1.
2. RHS transformation
   (a) Set $f_1 \leftarrow c$.
   (b) Modify the RHS: $f_i \leftarrow f_i - A_{i1} c, \quad i = 2, \ldots, p + 1$.

If the original matrix *is not symmetric*, only the steps (1a) and (2a) are required, since the other steps are only necessary to preserve the symmetry of the original matrix.

B.1.2 2D case

In that case, let us write the solution $u(x, y)$ as

$$u(x, y) = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} u_{ij} \Lambda_i(x) \Lambda_j(y), \tag{50}$$

where N_1, N_2 are the number of elements in each dimension. Assuming the BC $u(0,y) = g(y)$, and since $\Lambda_i(0) = \delta_{i1}$, the solutions u_{ij} should satisfy

$$\sum_{j=1}^{N_2} u_{1j}\Lambda_j(y) = g(y). \tag{51}$$

If $g(y)$ is constant $g(y) = c$, we obtain the trivial solution $u_{1,j} = c$ since $\sum_{j=1}^{N_2} \Lambda_j(y) = 1$ [1]. For non-uniform g, at least 2 methods can be used to obtain the N_2 unknowns u_{1j} satisfying the equation above:

1. By *collocating* Eq.(51) on a *suitable* set of points $[y_k]_{k=1}^{N_2}$, the problem is reduced to an *interpolation* one (see section "Spline Interpolation" in [1]).

2. By *minimizing* the residual norm of Eq.(51) defined as follows:

$$R = \left\| \sum_{j=1}^{N_2} c_j\Lambda_j(y) - g(y) \right\|^2 = \int dy \left\{ \left[\sum_{j=1}^{N_2} c_j\Lambda_j(y) \right]^2 - 2g(y) \sum_{j=1}^{N_2} c_j\Lambda_j(y) + g^2(y) \right\} \tag{52}$$

$$\frac{\partial R}{\partial c_k} = 2\int dy \left[\sum_{j=1}^{N_2} c_j\Lambda_j(y)\Lambda_k(y) - g(y)\Lambda_k(y) \right] = 0, \quad k = 1, \ldots, N_2, \tag{53}$$

the boundary solutions c_j can be calculated by solving the following *weak-form* of Eq.(51):

$$\sum_{j=1}^{N_2} c_j \int dy\Lambda_j(y)\Lambda_k(y) = \int dy\Lambda_k(y)g(y), \qquad k = 1, \ldots, N_2. \tag{54}$$

Once the values of c_j known, the procedure described for the 1D case above can be applied to satisfy each of the N_2 conditions $u_{1j} = c_j$.

A full example for solving the cylindrical Laplace equation in cylindrical coordinates:

$$\begin{gathered} \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \phi}{\partial r} \right) + \frac{1}{r^2}\frac{\partial^2 \phi}{\partial \theta^2} = 0 \\ \phi(r = 1, \theta) = \cos m\theta. \end{gathered} \tag{55}$$

is given in `bpslines/examples/dirichlet/poisson.f90`.

B.2 Unicity on the axis

Denoting the N solutions at the axis by $(u_1, \ldots, u_N)$, and their transforms by $(\hat{u}_1, \ldots, \hat{u}_N)$ defined by

$$\begin{array}{c} u_1 - u_N = \hat{u}_1 \\ u_2 - u_N = \hat{u}_2 \\ \vdots \\ u_{N-1} - u_N = \hat{u}_{N-1} \\ u_N = \hat{u}_N \end{array} \quad \Longrightarrow \quad \begin{array}{c} u_1 = \hat{u}_1 + \hat{u}_N \\ u_2 = \hat{u}_2 + \hat{u}_N \\ \vdots \\ u_{N-1} = \hat{u}_{N-1} + \hat{u}_N \\ u_N = \hat{u}_N, \end{array} \tag{56}$$

the unicity condition can be specified by simply imposing

$$\hat{u}_1 = \hat{u}_2 = \ldots = \hat{u}_{N-1} = 0. \tag{57}$$

From (56), the *transformation matrix* $\mathbf{U}$ is defined as

$$\mathbf{u} = \mathbf{U} \cdot \hat{\mathbf{u}}, \qquad \mathbf{U} = \begin{pmatrix} 1 & 0 & \dots & 0 & 1 \\ 0 & 1 & \dots & 0 & 1 \\ & & \ddots & & \vdots \\ 0 & 0 & \dots & 1 & 1 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}, \quad \mathbf{U^T} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ & & \ddots & & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 1 & 1 & \dots & 1 & 1 \end{pmatrix}. \tag{58}$$

Matrix product $\mathbf{A} \cdot \mathbf{U}$

$$\mathbf{A} \cdot \mathbf{U} = \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,N-1} & \sum_j A_{1,j} \\ A_{2,1} & A_{2,2} & \dots & A_{2,N-1} & \sum_j A_{2,j} \\ & & \ddots & & \vdots \\ A_{N-1,1} & A_{N-1,2} & \dots & A_{N-1,N-1} & \sum_j A_{N-1,j} \\ A_{N,1} & A_{N,2} & \dots & A_{N,N-1} & \sum_j A_{N,j} \end{pmatrix}. \tag{59}$$

Thus *right multiply by* $\mathbf{U}$ is equivalent to put the *the sum of each row on the last column.*

Matrix product $\mathbf{U^T} \cdot \mathbf{A}$

$$\mathbf{U^T} \cdot \mathbf{A} = \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,N-1} & A_{1,N} \\ A_{2,1} & A_{2,2} & \dots & A_{2,N-1} & A_{2,N} \\ & & \ddots & & \vdots \\ A_{N-1,1} & A_{N-1,2} & \dots & A_{N-1,N-1} & A_{N-1,N} \\ \sum_i A_{i,1} & \sum_i A_{i,2} & \dots & \sum_i A_{i,N-1} & \sum_i A_{i,N} \end{pmatrix}. \tag{60}$$

Thus *left multiply by* $\hat{\mathbf{U}}$ is equivalent to put the *the sum of each column on the last row.*

Product $\hat{\mathbf{U}} \cdot \mathbf{b}$

$$\hat{\mathbf{b}} = \mathbf{U^T} \cdot \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ \sum_i b_i \end{pmatrix}, \tag{61}$$

Transformation of the original matrix equation The full original linear system, obtained from the discretization of the 2D r, θ polar coordinates can be written as:

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix}, \tag{62}$$

where the solution array is split into the solutions $\mathbf{u}$ at $r = 0$ and the solutions $\mathbf{v}$ on the remaining domain. The transformed system can thus be written as

$$\begin{pmatrix} \mathbf{U^T} & 0 \\ 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \begin{pmatrix} \mathbf{U} & 0 \\ 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{u}} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{U^T} & 0 \\ 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix},$$

$$\implies \begin{pmatrix} \mathbf{U^T A U} & \mathbf{U^T B} \\ \mathbf{C U} & \mathbf{D} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{u}} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{U^T b} \\ \mathbf{c} \end{pmatrix}, \tag{63}$$

Notice that the transformation preserves any symmetry existing in the original system (62). The transformed matrix is finally given in the following where only the modified elements are shown and the sum is only over the first N points in θ direction. The $\times$ symbol denotes unmodified elements.

$$\begin{pmatrix} \times & \times & \times & \times & \sum_j A_{1,j} & \times & \times \\ \times & \times & \times & \times & \sum_j A_{2,j} & \times & \times \\ \times & \times & \times & \times & \vdots & \times & \times \\ \times & \times & \times & \times & \sum_j A_{N-1,j} & \times & \times \\ \sum_i A_{i,1} & \sum_i A_{i,2} & \dots & \sum_i A_{i,N-1} & \sum_{i,j} A_{i,j} & \sum_i A_{i,N+1} & \dots \\ \times & \times & \times & \times & \sum_j A_{N+1,j} & \times & \times \\ \times & \times & \times & \times & \vdots & \times & \times \end{pmatrix} \tag{64}$$

Only the N^{th} column and the N^{th} row are affected by the transformation. Applying now the unicity condition (57) the final transformed system reads:

$$\begin{pmatrix} 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & \vdots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & \sum_{i,j} A_{i,j} & \sum_i A_{i,N+1} & \dots \\ 0 & 0 & \dots & 0 & \sum_j A_{N+1,j} & \times & \times \\ 0 & 0 & \dots & 0 & \vdots & \times & \times \end{pmatrix} \begin{pmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \vdots \\ \hat{u}_{N-1} \\ \hat{u}_N \\ u_{N+1} \\ \vdots \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sum_i b_i \\ b_{N+1} \\ \vdots \end{pmatrix}. \tag{65}$$

C MATRIX Reference

The following conventions are adopted in the routine descriptions:

- `[z]` means optional: for example `TYPE([z]gemat)` declares a variable which can be of type `gemat` or `zgemat`.
- The symbol "|" is the logical `OR` operator. Thus

  ```
  TYPE([z]gemat|[z]gbmat) :: mat
  ```

  declares that `mat` can be either of type `gemat`, `zgemat`, `pbmat` or `zpbmat`.
- In a same declaration block, if a scalar or array of type `DOUBLE PRECISION|COMPLEX` is declared together with a matrix object which can be also complex, both should be either real or complex. For example in the routine `updtmat`, if `mat` of type `zgbmat`, `val` should be complex.

C.1 init

```
TYPE([z]gemat) :: mat
  SUBROUTINE init(n, nterms, mat ,kmat)
TYPE([z]gbmat) :: mat
  SUBROUTINE init(kl, ku, n, nterms, mat, kmat)
TYPE([z]pbmat) :: mat
  SUBROUTINE init(ku, n, nterms, mat, kmat)
TYPE([z]periodic_mat) :: mat
  SUBROUTINE init(kl, ku, n, nterms, mat, kmat)
```

```
INTEGER, INTENT(in) :: kl, ku, n, nterms
INTEGER, OPTIONAL   :: kmat
```

Purpose: Initialize data structure for matrix

Arguments:

```
n            : rank of matrix
kl, ku       : number of sub and super diagonals
nterms       : number of terms in weak form
kmat         : matrix id
mat          : matrix object
```

C.2 destroy

```
SUBROUTINE destroy(mat)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
```

Purpose: Free matrix memory

Arguments:

```
mat         : matrix object
```

C.3 updmat

```
SUBROUTINE updtmat(mat, i, j, val)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: val
```

Purpose: Update (accumulate) element A_{ij}

Arguments:

```
mat         : matrix object
i           : row index
j           : column index
val         : input value
```

C.4 putele

```
SUBROUTINE putele(mat, i, j, val)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: val
```

Purpose: Overwrite element A_{ij}

Arguments:

```
mat         : matrix object
i           : row index
j           : column index
val         : input value
```

C.5 putrow

```
SUBROUTINE putrow(mat, i, arr)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: i
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: arr(:)
```

Purpose: Overwrite a matrix row

Arguments:

```
mat          : matrix object
i            : row index
arr          : input array
```

C.6 putcol

```
SUBROUTINE putcol(mat, j, arr)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: arr(:)
```

Purpose: Overwrite a matrix row

Arguments:

```
mat          : matrix object
j            : column index
arr          : input array
```

C.7 getele

```
SUBROUTINE getele(mat, i, j, val)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: val
```

Purpose: Get element A_{ij}

Arguments:

```
mat          : matrix object
i            : row index
j            : column index
val          : output value
```

C.8 getrow

```
SUBROUTINE getrow(mat, i, arr)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: i
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: arr(:)
```

Purpose: Get a matrix row

Arguments:

```
mat         : matrix object
i           : row index
arr         : output array
```

C.9 getcol

```
SUBROUTINE getcol(mat, j, arr)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  INTEGER, INTENT(IN) :: j
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: arr(:)
```

Purpose: Get a matrix column

Arguments:

```
mat         : matrix object
i           : column index
arr         : output array
```

C.10 vmx

```
FUNCTION vmx(mat, x)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  DOUBLE PRECISION|COMPLEX, DIMENSION(:), INTENT(in) :: x
  DOUBLE PRECISION|COMPLEX, DIMENSION(SIZE(x)) :: vmx
```

Purpose: Matrix-vector product Ax

Arguments:

```
mat         : matrix object
x           : input array
vmx         : output array
```

C.11 mcopy

```
SUBROUTINE mcopy(mata, matb)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mata, matb
```

Purpose: Matrix copy: $B = A$

Arguments:

```
mata        : input matrix object
matb        : output matrix object
```

C.12 maddto

```
SUBROUTINE maddto(mata, alpha, matb)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mata, matb
  DOUBLE PRECISION|COMPLEX : alpha
```

Purpose: Matrix addition: $A \leftarrow A + \alpha B$

Arguments:

```
      mata        : input matrix object
      matb        : output matrix object
     alpha        : input scalar
```

C.13 determinant

```
SUBROUTINE determinant(mat, base, pow)
  TYPE([z]gemat|[z]gbmat|[z]pbmat) :: mat
  INTEGER, INTENT(out) :: pow
  DOUBLE PRECISION|COMPLEX : base
```

Purpose: Returns the determinant of matrix as $D = \text{base} \times 10^{\text{pow}}$

Arguments:

```
      mat         : input matrix object
      base        : mantissa of determinant
      pow         : exponent of determinant
```

C.14 factor

```
SUBROUTINE factor(mat)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
```

Purpose: LU (Cholesky for symmetric/hermitian matrix) factorization

Arguments:

```
      mat         : inout matrix object
```

C.15 bsolve

```
SUBROUTINE bsolve(mat)
  TYPE([z]gemat|[z]gbmat|[z]pbmat|[z]periodic_mat) :: mat
  DOUBLE PRECISION|COMPLEX, DIMENSION [(:)] :: rhs
  DOUBLE PRECISION|COMPLEX, DIMENSION [(:),] OPTIONAL, INTENT (out) :: sol
```

Purpose: Solve the linear system using the factored matrix, for a single or multiple right-hand-side

Arguments:

```
      mat  : input factored matrix object
      rhs  : input right-hand-side, overwriten by the solution if sol is not present
      sol  : contains solution
```

D SPMAT Reference

The following conventions are adopted in the routine descriptions:

- `[z]` means optional: for example `TYPE([z]gemat)` declares a variable which can be of type `gemat` or `zgemat`.
- The symbol "|" is the logical `OR` operator. Thus

  ```
  TYPE([z]gemat|[z]gbmat) :: mat
  ```

  declares that `mat` can be either of type `gemat`, `zgemat`, `pbmat` or `zpbmat`.
- In a same declaration block, if a scalar or array of type `DOUBLE PRECISION|COMPLEX` is declared together with a matrix object which can be also complex, both should be either real or complex. For example in the routine `updtmat`, if `mat` of type `zgbmat`, `val` should be complex.

D.1 init

```
SUBROUTINE init(n, mat, istart, iend)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(in), OPTIONAL :: istart, iend
  INTEGER, INTENT(in) :: n
```

Purpose: Initialize an empty sparse matrix of n rows.

Arguments:

```
n            : rank of matrix
mat          : matrix object
istart, iend : range of row indices. By default istart=1, iend=n
```

D.2 destroy

```
SUBROUTINE destroy(mat)
  TYPE([z]spmat) :: mat
```

Purpose: Free matrix memory

Arguments:

```
mat         : matrix object
```

D.3 updmat

```
SUBROUTINE updtmat(mat, i, j, val)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: val
```

Purpose: Update (accumulate) an existing element A_{ij} or insert it in the linked list in an increasing order in the column index j.

Arguments:

```
mat         : matrix object
i           : row index
j           : column index
val         : input value
```

D.4 putele

```
SUBROUTINE putele(mat, i, j, val, nlforce_zero)
  TYPE([z]pbmat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: val
  LOGICAL, INTENT(in), OPTIONAL :: nlforce_zero
```

Purpose: Overwrite an existing element A_{ij} or insert it in the linked list in an increasing order in the column index j.

Arguments:

```
mat          : matrix object
i            : row index
j            : column index
val          : input value
nlforce_zero : Never remove an existing element when input is zero if TRUE
               FALSE by default
```

D.5 putrow

```
SUBROUTINE putrow(mat, i, arr, col, nlforce_zero)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: i
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: arr(:)
  INTEGER, INTENT(in), OPTIONAL :: col(:)
  LOGICAL, INTENT(in), OPTIONAL :: nlforce_zero
```

Purpose: Overwrite a matrix row

Arguments:

```
mat          : matrix object
i            : row index
arr          : input array
col          : input array containing column indices
nlforce_zero : Never remove an existing element when input is zero if TRUE
               FALSE by default
```

D.6 putcol

```
SUBROUTINE putcol(mat, j, arr, nlforce_zero)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: j
  DOUBLE PRECISION|COMPLEX, INTENT(IN) :: arr(:)
```

Purpose: Overwrite a matrix row

Arguments:

```
mat          : matrix object
j            : column index
arr          : input array
nlforce_zero : Never remove an existing non-zero element if .TRUE.
               .FALSE. by default
```

D.7 getele

```
SUBROUTINE getele(mat, i, j, val)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: i, j
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: val
```

Purpose: Get element A_{ij}

Arguments:

```
mat         : matrix object
i           : row index
j           : column index
val         : output value
```

D.8 getrow

```
SUBROUTINE getrow(mat, i, arr, col)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: i
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: arr(:)
  INTEGER, INTENT(out), OPTIONAL :: col(:)
```

Purpose: Get a matrix row and optionally the column indices

Arguments:

```
mat         : matrix object
i           : row index
arr         : output array
col         : output array containing column indices
```

D.9 getcol

```
SUBROUTINE getcol(mat, j, arr)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(IN) :: j
  DOUBLE PRECISION|COMPLEX, INTENT(OUT) :: arr(:)
```

Purpose: Get a matrix column

Arguments:

```
mat         : matrix object
i           : column index
arr         : output array
```

D.10 get_count

```
INTEGER FUNCTION get_count(mat, nnz)
  TYPE([z]spmat) :: mat
  INTEGER, INTENT(out), OPTIONAL :: nnz(:)
```

Purpose: Returns the number of non-zeros and optionally an array of numbers of non-zeros on each row

Arguments:

```
mat         : matrix object
nnz         : array containing numbers of non-zeros on each row.
```

E PARDISO_BSPLINES Reference

The subroutines `updmat`, `putele`, `putrow`, `putcol`, `getele`, `getrow`, `getcol`, `vmx`, `mcopy`, `maddto` and `destroy` have *exactly* the same list of arguments as those from the `MATRIX` module (as documented in Appendix C), except for the matrix types. Below, we show only the routines that have different arguments. The same conventions as before are used for the routine description.

E.1 init

```
SUBROUTINE init(n, nterms, mat, kmat, nlsym, [nlherm,] nlpos, &
     &                      nlforce_zero)
  INTEGER, INTENT(in)           :: n, nterms
  TYPE([z]pardiso_mat) :: mat
  INTEGER, OPTIONAL, INTENT(in) :: kmat
  LOGICAL, OPTIONAL, INTENT(in) :: nlsym
  LOGICAL, OPTIONAL, INTENT(in) :: nlpos
  LOGICAL, OPTIONAL, INTENT(in) :: nlforce_zero
```

Purpose: Initialize the PARDISO solver. A SPMAT matrix of n empty rows is initialized.

Arguments:

```
n            : rank of matrix
nterms       : number of terms in weak form
kmat         : matrix id
mat          : matrix object
nlsym        : symmetric or not. Default is .FALSE.
nlherm       : Hermitian or not for complex matrix . Default is .FALSE.
nlpos        : Positive-definite or not. Default is .TRUE.
nlforce_zero : Never remove an existing non-zero element if .TRUE.
               .TRUE. by default
```

E.2 clear_mat

```
SUBROUTINE clear_mat(mat)
  TYPE([z]pardiso_mat)         :: mat
```

Purpose: Clear matrix, keeping its sparse structure unchanged

Arguments:

```
mat          : matrix object
```

E.3 psum_mat

```
SUBROUTINE sum_mat(mat, comm)
  TYPE([z]pardiso_mat)         :: mat
  INTEGER, INTENT(in)          :: comm
```

Purpose: Parallel sum of matrices. Result matrix is placed in the sparse matrix mat%mat on all processes of comm.

Arguments:

```
mat          : matrix object
comm         : communicator
```

E.4 p2p_mat

```
SUBROUTINE p2p_mat(mat, dest, extyp, op, comm)
  TYPE([z]pardiso_mat)         :: mat
  INTEGER, INTENT(in)          :: dest
  CHARACTER(len=*), INTENT(in) :: extyp ! ('send', 'recv', 'sendrecv')
  CHARACTER(len=*), INTENT(in) :: op    ! ('put', 'updt')
  INTEGER, INTENT(in)          :: comm
```

Purpose: Point-to-point combine sparse matrix between 2 processes.

Arguments:

```
mat          : matrix object
dest         : rank of remote process
extyp        : exchange type ('send', 'recv', 'sendrecv')
op           : operation type ('put', 'updt')
comm         : communicator
```

E.5 get_count

```
INTEGER FUNCTION get_count(mat, nnz)
  TYPE([z]pardiso_mat) :: mat
  INTEGER, INTENT(out), OPTIONAL :: nnz(:)
```

Purpose: Returns the number of non-zeros and optionally an array of numbers of non-zeros on each row

Arguments:

```
mat         : matrix object
nnz         : array containing numbers of non-zeros on each row.
```

E.6 factor

```
SUBROUTINE factor(mat, nlreord, nlmetis, debug)
  TYPE([z]pardiso_mat)          :: mat
  LOGICAL, OPTIONAL, INTENT(in) :: nlreord
  LOGICAL, OPTIONAL, INTENT(in) :: nlmetis
  LOGICAL, OPTIONAL, INTENT(in) :: debug
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat         : matrix object
nlreord     : call reord_mat if .TRUE. (default is .TRUE.)
nlmetis     : use METIS nested dissection for reoredering. Default
              is minimum degree alogorithm.
debug       : verbose output from PARDISO if .TRUE. Default is .FALSE.
```

E.7 bsolve

```
SUBROUTINE bsolve_pardiso_mat1(mat, rhs, sol, nref, debug)
  TYPE([z]pardiso_mat)               :: mat
  DOUBLE PRECISION|COMPLEX           :: rhs(:)
  DOUBLE PRECISION|COMPLEX, OPTIONAL :: sol(:)
  INTEGER, OPTIONAL                  :: nref
  LOGICAL, OPTIONAL, INTENT(in)      :: debug
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat   : matrix object
rhs   : input right-hand-side, overwriten by the solution if sol is not present
sol   : contains solution
ref   : maximum number of refinement steps. Default is 0 (no refinement).
debug : verbose output from PARDISO if .TRUE. Default is .FALSE.
```

E.8 to_mat

```
SUBROUTINE to_mat(mat)
  TYPE([z]pardiso_mat)          :: mat
```

Purpose: Convert linked list spmat to pardiso matrix structure

Arguments:

```
mat         : matrix object
```

E.9 reord_mat

```
SUBROUTINE reord_mat(mat, nlmetis, debug)
  TYPE([z]pardiso_mat)          :: mat
  LOGICAL, OPTIONAL, INTENT(in) :: nlmetis
  LOGICAL, OPTIONAL, INTENT(in) :: debug
```

Purpose: Reordering and symbolic factorization

Arguments:

```
mat         : matrix object
nlmetis     : use METIS nested dissection for reoredering. Default
              is minimum degree alogorithm.
debug       : verbose output from PARDISO if .TRUE. Default is .FALSE.
```

E.10 numfact

```
SUBROUTINE numfact(mat, debug)
  TYPE([z]pardiso_mat)          :: mat
  LOGICAL, OPTIONAL, INTENT(in) :: debug
```

Purpose: Numerical factorization

Arguments:

```
mat         : matrix object
debug       : verbose output from PARDISO if .TRUE. Default is .FALSE.
```

F [P]WSMP_BSPLINES Reference

The subroutines `updmat`, `putele`, `putrow`, `putcol`, `getele`, `getrow`, `getcol`, `vmx`, `mcopy`, `maddto` and `destroy` have *exactly* the same list of arguments as those from the `MATRIX` module (as documented in Appendix C), except for the matrix types. Below, we show only the routines that have different arguments. The same conventions as before are used for the routine description.

F.1 init

```
SUBROUTINE init(n, nterms, mat, kmat, nlsym, [nlherm,] nlpos, &
     &                      nlforce_zero, [comm_in])
  INTEGER, INTENT(in)           :: n, nterms
  TYPE([z]wsmp_mat) :: mat
  INTEGER, OPTIONAL, INTENT(in) :: kmat
  LOGICAL, OPTIONAL, INTENT(in) :: nlsym
  LOGICAL, OPTIONAL, INTENT(in) :: nlpos
  LOGICAL, OPTIONAL, INTENT(in) :: nlforce_zero
  INTEGER, OPTIONAL, INTENT(in) :: comm_in
```

Purpose: Initialize the WSMP solver. A SPMAT matrix of n empty rows is initialized.

Arguments:

```
n            : rank of matrix
nterms       : number of terms in weak form
kmat         : matrix id
mat          : matrix object
nlsym        : symmetric or not. Default is .FALSE.
nlherm       : Hermitian or not for complex matrix . Default is .FALSE.
nlpos        : Positive-definite or not. Default is .TRUE.
nlforce_zero : Never remove an existing non-zero element if .TRUE.
               .TRUE. by default
comm_in      : MPI communicator. By default MPI_COMM_WORLD (only in PWSMP_BSPLINES)
```

F.2 clear_mat

```
SUBROUTINE clear_mat(mat)
  TYPE([z]wsmp_mat) :: mat
```

Purpose: Clear matrix, keeping its sparse structure unchanged

Arguments:

```
mat          : matrix object
```

F.3 psum_mat

```
SUBROUTINE sum_mat(mat, comm)
  TYPE([z]wsmp_mat)         :: mat
  INTEGER, INTENT(in)          :: comm
```

Purpose: Parallel sum of matrices. Result matrix is placed in the sparse matrix mat%mat on all processes of comm.

Arguments:

```
mat          : matrix object
comm         : communicator
```

F.4 p2p_mat

```
SUBROUTINE p2p_mat(mat, dest, extyp, op, comm)
  TYPE([z]wsmp_mat)         :: mat
  INTEGER, INTENT(in)          :: dest
  CHARACTER(len=*), INTENT(in) :: extyp ! ('send', 'recv', 'sendrecv')
  CHARACTER(len=*), INTENT(in) :: op    ! ('put', 'updt')
  INTEGER, INTENT(in)          :: comm
```

Purpose: Point-to-point combine sparse matrix between 2 processes.

Arguments:

```
mat          : matrix object
dest         : rank of remote process
extyp        : exchange type ('send', 'recv', 'sendrecv')
op           : operation type ('put', 'updt')
comm         : communicator
```

F.5 get_count

```
INTEGER FUNCTION get_count(mat, nnz)
  TYPE([z]wsmp_mat) :: mat
  INTEGER, INTENT(out), OPTIONAL :: nnz(:)
```

Purpose: Returns the number of non-zeros and optionally an array of numbers of non-zeros on each row

Arguments:

```
mat         : matrix object
nnz         : array containing numbers of non-zeros on each row.
```

F.6 factor

```
SUBROUTINE factor(mat, nlreord)
  TYPE([z]wsmp_mat)          :: mat
  LOGICAL, OPTIONAL, INTENT(in) :: nlreord
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat         : matrix object
nlreord     : call reord_mat if .TRUE. (default is .TRUE.)
```

F.7 bsolve

```
SUBROUTINE bsolve_wsmp_mat1(mat, rhs, sol, nref)
  TYPE([z]wsmp_mat)               :: mat
  DOUBLE PRECISION|COMPLEX           :: rhs(:)
  DOUBLE PRECISION|COMPLEX, OPTIONAL :: sol(:)
  INTEGER, OPTIONAL                  :: nref
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat   : matrix object
rhs   : input right-hand-side, overwriten by the solution if sol is not present
sol   : contains solution
ref   : maximum number of refinement steps. Default is 0 (no refinement).
debug : verbose output from WSMP if .TRUE. Default is .FALSE.
```

F.8 to_mat

```
SUBROUTINE to_mat(mat)
  TYPE([z]wsmp_mat)          :: mat
```

Purpose: Convert linked list spmat to wsmp matrix structure

Arguments:

```
mat         : matrix object
```

F.9 reord_mat

```
SUBROUTINE reord_mat(mat)
  TYPE([z]wsmp_mat)          :: mat
```

Purpose: Reordering and symbolic factorization

Arguments:

```
mat         : matrix object
```

F.10 numfact

```
SUBROUTINE numfact(mat)
  TYPE([z]wsmp_mat)          :: mat
```

Purpose: Numerical factorization

Arguments:

```
mat         : matrix object
```

G MUMPS_BSPLINES Reference

The subroutines `updmat`, `putele`, `putrow`, `putcol`, `getele`, `getrow`, `getcol`, `vmx`, `mcopy`, `maddto` and `destroy` have *exactly* the same list of arguments as those from the `MATRIX` module (as documented in Appendix C), except for the matrix types. Below, we show only the routines that have different arguments. The same conventions as before are used for the routine description.

G.1 init

```
SUBROUTINE init(n, nterms, mat, kmat, nlsym, [nlherm,] nlpos, &
     &                      nlforce_zero, comm_in)
  INTEGER, INTENT(in)           :: n, nterms
  TYPE([z]mumps_mat) :: mat
  INTEGER, OPTIONAL, INTENT(in) :: kmat
  LOGICAL, OPTIONAL, INTENT(in) :: nlsym
  LOGICAL, OPTIONAL, INTENT(in) :: nlpos
  LOGICAL, OPTIONAL, INTENT(in) :: nlforce_zero
  INTEGER, OPTIONAL, INTENT(in) :: comm_in
```

Purpose: Initialize the MUMPS solver. A SPMAT matrix of n empty rows is initialized.

Arguments:

```
n            : rank of matrix
nterms       : number of terms in weak form
kmat         : matrix id
mat          : matrix object
nlsym        : symmetric or not. Default is .FALSE.
nlherm       : Hermitian or not for complex matrix . Default is .FALSE.
nlpos        : Positive-definite or not. Default is .TRUE.
nlforce_zero : Never remove an existing non-zero element if .TRUE.
               .TRUE. by default
comm_in      : MPI communicator. By default MPI_COMM_SELF (serial mode).
```

G.2 clear_mat

```
SUBROUTINE clear_mat(mat)
  TYPE([z]mumps_mat) :: mat
```

Purpose: Clear matrix, keeping its sparse structure unchanged

Arguments:

```
mat          : matrix object
```

G.3 psum_mat

```
SUBROUTINE sum_mat(mat, comm)
  TYPE([z]mumps_mat)         :: mat
  INTEGER, INTENT(in)          :: comm
```

Purpose: Parallel sum of matrices. Result matrix is placed in the sparse matrix mat%mat on all processes of comm.

Arguments:

```
mat          : matrix object
comm         : communicator
```

G.4 p2p_mat

```
SUBROUTINE p2p_mat(mat, dest, extyp, op, comm)
  TYPE([z]mumps_mat)         :: mat
  INTEGER, INTENT(in)          :: dest
  CHARACTER(len=*), INTENT(in) :: extyp ! ('send', 'recv', 'sendrecv')
  CHARACTER(len=*), INTENT(in) :: op    ! ('put', 'updt')
  INTEGER, INTENT(in)          :: comm
```

Purpose: Point-to-point combine sparse matrix between 2 processes.

Arguments:

```
mat          : matrix object
dest         : rank of remote process
extyp        : exchange type ('send', 'recv', 'sendrecv')
op           : operation type ('put', 'updt')
comm         : communicator
```

G.5 get_count

```
INTEGER FUNCTION get_count(mat, nnz)
  TYPE([z]mumps_mat) :: mat
  INTEGER, INTENT(out), OPTIONAL :: nnz(:)
```

Purpose: Returns the number of non-zeros and optionally an array of numbers of non-zeros on each row

Arguments:

```
mat         : matrix object
nnz         : array containing numbers of non-zeros on each row.
```

G.6 factor

```
SUBROUTINE factor(mat, nlreord)
  TYPE([z]mumps_mat)          :: mat
  LOGICAL, OPTIONAL, INTENT(in) :: nlreord
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat         : matrix object
nlreord     : call reord_mat if .TRUE. (default is .TRUE.)
```

G.7 bsolve

```
SUBROUTINE bsolve_mumps_mat1(mat, rhs, sol, nref)
  TYPE([z]mumps_mat)               :: mat
  DOUBLE PRECISION|COMPLEX           :: rhs(:)
  DOUBLE PRECISION|COMPLEX, OPTIONAL :: sol(:)
  INTEGER, OPTIONAL                  :: nref
```

Purpose: Wrapper of to_mat, reord_mat and numfact

Arguments:

```
mat   : matrix object
rhs   : input right-hand-side, overwriten by the solution if sol is not present
sol   : contains solution
ref   : maximum number of refinement steps. Default is 0 (no refinement).
debug : verbose output from MUMPS if .TRUE. Default is .FALSE.
```

G.8 to_mat

```
SUBROUTINE to_mat(mat)
  TYPE([z]mumps_mat)          :: mat
```

Purpose: Convert linked list spmat to mumps matrix structure

Arguments:

```
mat         : matrix object
```

G.9 reord_mat

```
SUBROUTINE reord_mat(mat)
  TYPE([z]mumps_mat)          :: mat
```

Purpose: Reordering and symbolic factorization

Arguments:

```
mat         : matrix object
```

G.10 numfact

```
SUBROUTINE numfact(mat)
  TYPE([z]mumps_mat)          :: mat
```

Purpose: Numerical factorization

Arguments:

```
mat         : matrix object
```

References

[1] BSPLINES Reference Guide.

[2] http://www.pardiso-project.org/

[3] http://www-users.cs.umn.edu/~agupta/wsmp.html

[4] http://graal.ens-lyon.fr/MUMPS/

[5] B. F. McMillan, et. al. *Rapid Fourier space solution of linear partial integro-differential equations in toroidal magnetic confinement geometries*, Computer Physics Communications 181(4), 715-719 (2010)