
PSBLAS-KINSOL interface

*Istituto per le Applicazioni del Calcolo “M. Picone”,
Consiglio Nazionale delle Ricerche
Pasqua D’Ambra
Fabio Durastante
Salvatore Filippone
PSBLAS 3.6.1 — Interface Version 1*

April 18, 2020

Contents

1	The logic behind a PSBLAS instrumented application	2
2	The NVECTOR_PSBLAS implementation	4
2.1	NVECTOR_PSBLAS accessor macros	4
2.2	NVECTOR_PSBLAS functions	5
3	The SUNMATRIX_PSBLAS implementation	10
3.1	SUNMATRIX_PSBLAS accessor macros	10
3.2	SUNMATRIX_PSBLAS functions	11
4	The SUNLINSOL_PSBLAS implementation	13
4.1	SUNLINSOL_PSBLAS accessor macros	15
4.2	SUNLINSOL_PSBLAS functions	15
4.3	Algebraic Multigrid Preconditioners: the MLD2P4 package	17
A	Matrix assembly routine	18
	References	20

1 The logic behind a PSBLAS instrumented application

Library structure

```
kinsol-psblas
├── config
├── doc
├── examples
├── include
├── src
└── test
```

The main aim of the PSBLAS library is the parallel implementation of iterative solvers for sparse linear systems,

$$Ax = b, \quad A \in \mathbb{K}^{n \times n}, \quad \mathbb{K} = \mathbb{R}, \mathbb{C}, \quad (1)$$

through the distributed memory paradigm operating with message passing. The library includes all the needed routines for this task, e.g, functions for multiplying sparse matrices by dense matrices, for solving block diagonal systems with triangular diagonal entries or for preprocessing sparse matrices.

Owner computes rule

The pivotal choice to be made in this setting regards the distribution of the coefficient matrix A for the linear system (1). In PSBLAS this choice is based on the **owner computes rule**: each unknown is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations.

If A is obtained from the discretization of a Partial Differential Equation (PDE), this allocation strategy is equivalent to the choice of a partition of the mesh into *sub-domains*.

! → PSBLAS supports *any* distribution that keeps together the coefficients of each matrix row there are **no other** constraints on the variable assignment.

Any PSBLAS application will always start with the construction of the parallel environment, i.e., of an MPI (virtual) parallel machine, that we call here context by means of the `psb_c_init` function as

```
psb_i_t ictxt, iam, np;
ictxt = psb_c_init();
psb_c_info(ictxt, &iam, &np);
```

that creates a parallel environment on np processors $0, \dots, np-1$, of which we are process iam .

`psb_c_init` : this subroutine initializes the PSBLAS parallel environment, defining a virtual parallel machine, the value `ictxt` in out is the communication context identifying the virtual parallel machine. This is always a duplicate of `basetxt`, so that library communications are completely separated from other communication operations

Prototype `psb_i_t psb_c_init();`

`psb_c_info` : this subroutine returns information about the PSBLAS parallel environment, defining a virtual parallel machine that is identified by the value `ictxt`. The values on return are the identifier of current process in the PSBLAS virtual parallel machine (`iam`), and the number of processes in the PSBLAS virtual parallel machine (`np`)

Prototype `psb_i_t psb_c_info(psb_i_t ictxt, psb_i_t *iam, psb_i_t *np);`

The next step is represented by the need of subdividing the index space among processes, and this creates a mapping from the “global” numbering $1, \dots, n$ to a “local” numbering in each process. This means that each process i will own a certain subset $1, \dots, n_{row_i}$, each element of which corresponds to a certain element of $1 \dots n$.

Therefore, after the initialization the first step is to establish an index space, and this is done with a call to one of the variants of the `psb_cdall` function to allocate a descriptor object `psb_c_descriptor`:

`psb_c_cdall_vg` : the association between an index and a process is specified via an integer vector `vg[]`, each index $i \in \{1, \dots, ng\}$ is assigned to process `vg[i]`. The vector `vg[]` must be identical on all calling processes, and its entries have the ranges $(0, \dots, np-1)$ or $(1, \dots, np)$ according to the fact that `psb_c_set_index_base(0)` or `psb_c_set_index_base(1)` has been called at the beginning. The size `ng` is specified one can chose to use the entire vector `vg[]`, thus having `vg[ng]`.

Prototype `psb_c_cdall_vg(psb_l_t ng, psb_i_t *vg, psb_i_t ictxt, psb_c_descriptor *cdh);`

`psb_c_cdall_vl` : the association is done by specifying the list of indices `vl[nl]` assigned to the current process; thus, the global problem size `nl` is given by the range of the aggregate of the individual vectors `vl[]` specified in the calling processes. The subroutine will check how many times each entry in the global index space $(1, \dots, nl)$ is specified in the input lists `vl[]`, therefore it allows for the presence of overlap in the input, and checks for the “orphan” indices.

Prototype `psb_c_cdall_vl(psb_i_t nl, psb_l_t *vl, psb_i_t ictxt, psb_c_descriptor *cdh);`

`psb_c_cdall_nl` : produces a generalized block-row distribution of the number of indices belonging to the current process in which each process i gets assigned a consecutive chunk of `nl` global indices,

Prototype `psb_c_cdall_nl(psb_i_t nl, psb_i_t ictxt, psb_c_descriptor *cdh);`

for the case of a simple minded block distribution, i.e., the index space is first numbered sequentially in a standard way, then the corresponding vector is distributed according to a block distribution directive.

Example 1. We consider as example the finite difference discretization of the following boundary value problem

$$-\frac{b_1 \partial^2 u}{\partial x^2} - \frac{b_2 \partial^2 u}{\partial y^2} - \frac{b_3 \partial^2 u}{\partial z^2} + \frac{a_1 \partial u}{\partial x} + \frac{a_2 \partial u}{\partial y} + a_3 \frac{\partial u}{\partial z} = 0, \quad (2)$$

for $(x, y, z) \in [0, 1]^3$, with Dirichlet boundary conditions, on a uniform grid with `idim` node per size. All the allocation procedure can be expressed as

```
psb_c_descriptor *cdh;
psb_i_t idim, nb, nlr, nl;
psb_l_t i, ng, *vl, k;

cdh=psb_c_new_descriptor();
psb_c_set_index_base(0);

/* Simple minded BLOCK data distribution */
ng = ((psb_l_t) idim)*idim*idim;
nb = (ng+np-1)/np;
nl = nb;
if ( (ng -iam*nb) < nl) nl = ng -iam*nb;
fprintf(stderr, "%d: Input data %d %ld %d %d\n", iam, idim, ng, nb, nl);
if ((vl=malloc(nb*sizeof(psb_l_t)))==NULL) {
    fprintf(stderr, "On %d: malloc failure\n", iam);
    psb_c_abort(ictxt);
}
i = ((psb_l_t)iam) * nb;
for (k=0; k<nl; k++)
    vl[k] = i+k;
psb_c_cdall_vl(nl, vl, ictxt, cdh);
```

Listing 1: "Example of allocation procedure for a 3D block data distribution"

Now that the initial distribution of the index space has been performed, we need to allocate dense vectors and sparse matrices on such index space, and thus we define the complete topology of our computational problem. Since our task is to use the capabilities of SUNDIALS-KINSOL to solve for nonlinear problems, here lies the core of the interfacing between the two codes.

Before the termination of the program, *after* having freed/destroyed all the objects instantiated on the descriptor `cdh`, you need also to free the memory occupied by it, and the to terminate the parallel environment. This is achieved by means of the following routines.

`psb_c_cdfree` : frees a communication descriptor

Prototype `psb_int psb_c_cdfree(psb_c_descriptor *cdh);`

`psb_c_exit` : this subroutine exits from the PSBLAS parallel virtual machine, observe that this routine may be called even if a previous call to `psb_info` has returned with `iam=-1`.

Prototype `psb_int psb_c_exit(psb_int ictxt);`

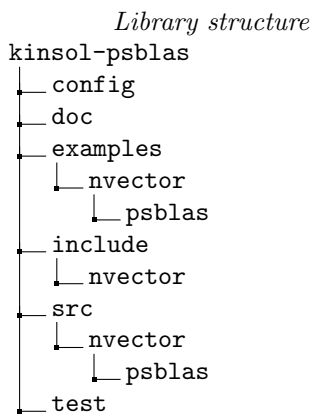
`psb_c_abort` : this subroutine aborts computation on the parallel virtual machine, and can be used when error that needs the program to fail occurs.

Prototype `psb_int psb_c_abort(psb_int ictxt);`

In the next two Sections 2, and Section 3 we describe such encapsulation for dense vector, and sparse matrices. Then in Section 4 we describe the interfacing with the **linear solvers** and **preconditioners**

Complete information on the PSBLAS data structures, and functions that are mentioned along the text can be found in [3].

2 The NVECTOR_PSBLAS implementation



The NVECTOR_PSBLAS implementation of the SUNDIALS NVECTOR module provides an interface to the PSBLAS code for handling distributed dense vectors. It defines the *content* field of `N_Vector` to be a structure containing the PSBLAS descriptor for the data distribution, a PSBLAS vector of double, and the PSBLAS communicator (context).

```

struct _N_VectorContent_PSBLAS {
    booleantype own_data; /*ownership of data*/
    psb_c_descriptor *cdh; /*descriptor for data distribution*/
    psb_c_dvector *pvec; /*PSBLAS vector*/
    int ictxt; /*PSBLAS communicator*/
};

```

All the vectors that have to interact needs to be instantiated on the same parallel context `ictxt`, and on the same data distribution `cdh`.

! →

The header file to include when using this module is `nvector_psbblas.h`. The installed module library to link to is `sundials_nvecpsblas.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

2.1 NVECTOR_PSBLAS accessor macros

The following macros are provided to access the content of a NVECTOR_PSBLAS vector. The suffix `_P` in the names denotes the fact that the data are in distributed memory.

```

#define NV_CONTENT_P(v)    ((N_VectorContent_PSBLAS)(v->
                           content))
#define NV_DESCRIPTOR_P(v) (NV_CONTENT_P(v)->cdh)
#define NV_OWNDATA_P(v)    (NV_CONTENT_P(v)->own_data)
#define NV_PVEC_P(v)       (NV_CONTENT_P(v)->pvec)
#define NV_IXTXT_P(v)      (NV_CONTENT_P(v)->ictxt)

```

NV_CONTENT_P(v) this macro gives access to the contents of the PSBLAS vector **N_Vector**.
 NV_DESCRIPTOR_P(v), these macros give instead individual access to the parts of the content of
 NV_OWNDATA_P(v), a PSBLAS parallel **N_Vector**.
 NV_PVEC_P(v) this macro provides the PSBLAS context used by the NVECTOR_PSBLAS
 NV_IXTXT_P(v) vectors.

2.2 NVECTOR_PSBLAS functions

The NVECTOR_PSBLAS implementation provides PSBLAS implementations of all the vectors operations listed in Tables 6.2, 6.3, and 6.4 of the original KINSOL library [1]. Following the standard nomenclature of the SUNDIALS library, their names are obtained from the ones listed in Tables 6.2, 6.3, and 6.4 by appending the suffix **_PSBLAS**. The NVECTOR_PSBLAS implementation provides the following additional user-callable routine:

N_VAsb_PSBLAS : This routine assemble the NVector after that all the elements have been inserted into it, i.e., after that all the calls to the **N_VMake_PSBLAS** routine have been completed. This is substantially a wrapper for the PSBLAS function `psb_c_dgeasb`.

Prototype `void N_VAsb_PSBLAS(N_Vector v)`

2.2.1 Description of the NVECTOR_PSBLAS functions

N_VNew_PSBLAS : This function creates and allocates memory for a parallel vector on the PSBLAS context `ictxt` with the communicator `cdh`

Prototype `N_Vector N_VNew_PSBLAS(int ictxt, psb_c_descriptor *cdh);`

N_VNewEmpty_PSBLAS : This function creates a new PSBLAS vector with empty data array.

Prototype `N_Vector N_VNewEmpty_PSBLAS(int ictxt, psb_c_descriptor *cdh);`

N_VMake_PSBLAS : Function to create a PSBLAS **N_Vector** with user data component. This function is substantially a wrapper for the PSBLAS function `psb_c_dgeins`.

Prototype `N_Vector N_VMake_PSBLAS(int ictxt, psb_c_descriptor *cdh, psb_int m, psb_int *irow, double *val);`

The PSBLAS context `ictxt` with the communicator `cdh` are the one defined for the whole programs, the integer `m` is the number of rows in `val[]` to be inserted, the array of integers `irow` is the indices of the rows to be inserted. Specifically, row `i` of `val` will be inserted into the local row corresponding to the global index row index `row[i]`.

! → This routine does not assemble the final vector. After the insertion of all the elements has been completed then the vector should be assembled by means of the **N_VAsb_PSBLAS** routine.

N_VCloneVectorArray_PSBLAS : This function creates an array of new parallel vectors (by cloning) an array of count parallel vectors `v`.

Prototype `N_Vector *N_VCloneVectorArray_PSBLAS(int count, N_Vector w)`

N_VCloneVectorArrayEmpty_PSBLAS : This function creates an array of count new parallel vectors with empty data array on the same communicator and context of the vector w.

Prototype `N_Vector *N_VCloneVectorArrayEmpty_PSBLAS(int count, N_Vector w)`

N_VDestroyVectorArray_PSBLAS : This function to frees an array of count N_Vectors created with N_VCloneVectorArray_PSBLAS

Prototype `void N_VDestroyVectorArray_PSBLAS(N_Vector *vs, int count)`

N_VGetLength_PSBLAS : This function returns the *global* vector length, this is substantially a wrapper for the PSBLAS function `psb_c_cd_get_global_rows`.

Prototype `sunindextype N_VGetLength_PSBLAS(N_Vector v)`

N_VGetLocalLength_PSBLAS : This function returns the *local* vector length, this is substantially a wrapper for the PSBLAS function `psb_c_cd_get_local_rows`.

Prototype `sunindextype N_VGetLocalLength_PSBLAS(N_Vector v)`

N_VPrint_PSBLAS : This function prints the local data in a parallel vector to stdout.

Prototype `void N_VPrint_PSBLAS(N_Vector x)`

N_VPrintFile_PSBLAS : This function prints the local data in a parallel vector to outfile.

Prototype `void N_VPrintFile_PSBLAS(N_Vector x, FILE* outfile)`

N_VGetVectorID_PSBLAS : This function returns the SUNDIALS identificative for the PSBLAS vector, since this is a custom implementation it returns the integer constant SUNDIALS_NVEC_CUSTOM.

Prototype `N_Vector_ID N_VGetVectorID_PSBLAS(N_Vector v)`

N_VCloneEmpty_PSBLAS : Clones a NVECTOR_PSBLAS with a NULL pvec field, and with value SUNFALSE in the own_data field.

Prototype `N_Vector N_VCloneEmpty_PSBLAS(N_Vector w)`

N_VClone_PSBLAS : Clones a NVECTOR_PSBLAS allocating its memory following the same communicator of the cloned one.

Prototype `N_Vector N_VClone_PSBLAS(N_Vector w)`

N_VDestroy_PSBLAS : Destroys a NVECTOR_PSBLAS freeing both the memory allocated for the corresponding PSBLAS vector, and the memory allocated for the NVECTOR_PSBLAS structure.

Prototype `void N_VDestroy_PSBLAS(N_Vector v)`

N_VSpace_PSBLAS : Returns storage requirements for one NVECTOR_PSBLAS. lrw contains the number of realtype words and liw contains the number of integer words.

Prototype `void N_VSpace_PSBLAS(N_Vector v, sunindextype *lrw, sunindextype *liw)`

N_VGetArrayPointer_PSBLAS : Returns a pointer to a realtype array from the NVECTOR_PSBLAS, this is the local portion of the distributed PSBLAS vector encapsulated in the N_Vector object.

Prototype `realtype *N_VGetArrayPointer_PSBLAS(N_Vector v)`

N_VSetArrayPointer_PSBLAS : This function is a dummy function, in PSBLAS we use **allocatable** objects for the local part of the distributed vector, and, moreover, we assume having an arbitrary distribution of the indexes.

Prototype	<code>void N_VSetArrayPointer_PSBLAS(psb.c.dvector *v_data, N_Vector v)</code>
<code>N_VLinearSum_PSBLAS</code> :	Performs the AXPBY BLAS operation between two NVECTOR_PSBLAS, the result can be both out-of- and in-place.
Prototype	<code>void N_VLinearSum_PSBLAS(realtype a, N_Vector x, realtype b, N_Vector y, N_Vector z)</code>
<code>N_VConst_PSBLAS</code> :	Sets all components of the NVECTOR_PSBLAS to a constant value and assembles it, the user does not need to assemble it.
Prototype	<code>void N_VConst_PSBLAS(realtype c, N_Vector z)</code>
<code>N_VProd_PSBLAS</code> :	Performs the entry-wise multiplication of two NVECTOR_PSBLAS, the result can be both out-of- and in-place.
Prototype	<code>void N_VProd_PSBLAS(N_Vector x, N_Vector y, N_Vector z)</code>
<code>N_VDiv_PSBLAS</code> :	Performs the entry-wise division of two NVECTOR_PSBLAS, the result can be both out-of- and in-place, it does not check for possible zero entries in the denominator. It is up to the user to guarantee that this does not happen.
Prototype	<code>void N_VDiv_PSBLAS(N_Vector x, N_Vector y, N_Vector z)</code>
<code>N_VScale_PSBLAS</code> :	Scales an NVECTOR_PSBLAS by a scalar c, the result can be both out-of- and in-place.
Prototype	<code>void N_VScale_PSBLAS(realtype c, N_Vector x, N_Vector z)</code>
<code>N_VAbs_PSBLAS</code> :	Sets the entries of an NVECTOR_PSBLAS to the absolute values of the entries of the input.
Prototype	<code>void N_VAbs_PSBLAS(N_Vector x, N_Vector z)</code>
<code>N_VInv_PSBLAS</code> :	Sets the entries of an NVECTOR_PSBLAS to the inverse of the entries of the input. It does not check for possible zero entries in the vector. It is up to the user to guarantee that this does not happen.
Prototype	<code>void N_VInv_PSBLAS(N_Vector x, N_Vector z)</code>
<code>N_VAddConst_PSBLAS</code> :	Adds a scalar to all components of an NVECTOR_PSBLAS and returns the result in another NVECTOR_PSBLAS object.
Prototype	<code>void N_VAddConst_PSBLAS(N_Vector x, realtype b, N_Vector z)</code>
<code>N_VDotProd_PSBLAS</code> :	Compute the dot product of two NVECTOR_PSBLAS objects.
Prototype	<code>realtype N_VDotProd_PSBLAS(N_Vector x, N_Vector y)</code>
<code>N_VMaxNorm_PSBLAS</code> :	Compute the max norm of an NVECTOR_PSBLAS object.
Prototype	<code>realtype N_VMaxNorm_PSBLAS(N_Vector x)</code>
<code>N_VWrmsNorm_PSBLAS</code> :	Compute the weighted (by the size) 2-norm of an NVECTOR_PSBLAS object.
Prototype	<code>realtype N_VWrmsNorm_PSBLAS(N_Vector x, N_Vector w)</code>
<code>N_VWrmsNormMask_PSBLAS</code> :	Returns the weighted root mean square norm of the NVECTOR_PSBLAS x with realtype weight vector w built using only the elements of x corresponding to positive elements of the NVECTOR_PSBLAS id.
Prototype	<code>realtype N_VWrmsNormMask_PSBLAS(N_Vector x, N_Vector w, N_Vector id)</code>

N_VMin_PSBLAS : Gives back the minimum entry of an NVECTOR_PSBLAS object.

Prototype `realtype N_VMin_PSBLAS(N_Vector x)`

N_VWL2Norm_PSBLAS : Returns the weighted Euclidean 2-norm of the NVECTOR_PSBLAS `x` with realtype weight vector `w`.

Prototype `realtype N_VWL2Norm_PSBLAS(N_Vector x, N_Vector w)`

N_VL1Norm_PSBLAS : Computes the 1-norm of an NVECTOR_PSBLAS object.

Prototype `realtype N_VL1Norm_PSBLAS(N_Vector x)`

N_VCompare_PSBLAS : Compares the components of the NVECTOR_PSBLAS `x` to the realtype scalar `c` and returns an NVECTOR_PSBLAS `z` such that

$$z_i = \begin{cases} 1.0, & |x_i| \geq c, \\ 0.0, & |x_i| \leq c. \end{cases}$$

Prototype `void N_VCompare_PSBLAS(realtype c, N_Vector x, N_Vector z)`

N_VInvTest_PSBLAS : Sets the entries of an NVECTOR_PSBLAS to the inverse of the entries of the input. It checks for possible zero entries in the vector. This routine returns a boolean assigned to `SUNTRUE` if all components of the vector are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Prototype `boolean_t N_VInvTest_PSBLAS(N_Vector x, N_Vector z)`

N_VConstrMask_PSBLAS : Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `SUNFALSE` if any element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Prototype `boolean_t N_VConstrMask_PSBLAS(N_Vector c, N_Vector x, N_Vector m)`

N_VMinQuotient_PSBLAS : This routine returns the minimum of the quotients obtained by term-wise dividing `num[i]` by `denom[i]`. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials/types.h`) is returned.

Prototype `realtype N_VMinQuotient_PSBLAS(N_Vector num, N_Vector denom)`

2.2.2 Fused operation

N_VLinearCombination_PSBLAS : This routine computes the linear combination of `nv` vectors with `n` elements

$$z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \dots, n-1$$

where `c` is an array of `nv` scalars, `X` is an array of NVECTOR_PSBLAS. When `z` is one of the vectors in `X`, then it is assumed to be the first vector in the vector array.

Prototype `int N_VLinearCombination_PSBLAS(int nvec, realtype* c, N_Vector* V, N_Vector z)`

N_VScaleAddMulti_PSBLAS : This routine scales and adds one vector to `nv` vectors with `n` elements:

$$z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \dots, n_v - 1, \quad i = 0, \dots, n - 1,$$

where c is an array of n_v scalars.

Prototype `int N_VScaleAddMulti_PSBLAS(int nvec, realtype* a, N_Vector x, N_Vector* Y, N_Vector* Z)`

`N_VDotProdMulti_PSBLAS` : This routine computes the dot product of a vector with n_v other vectors.

Prototype `int N_VDotProdMulti_PSBLAS(int nvec, N_Vector x, N_Vector *Y, realtype* dotprods)`

2.2.3 Vector array operations

`N_VLinearSumVectorArray_PSBLAS` : This routine computes the linear sum of two vector arrays containing n_v vectors of n elements:

$$z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1, \quad j = 0, \dots, n_v-1.$$

Prototype `int N_VLinearSumVectorArray_PSBLAS(int nvec, realtype a, N_Vector* X, realtype b, N_Vector* Y, N_Vector* Z)`

`N_VScaleVectorArray_PSBLAS` : This routine scales each vector of n elements in a vector array of n_v vectors by a (potentially different) constant:

$$z_{j,i} = c_j x_{j,i}, \quad i = 0, \dots, n-1, \quad j = 0, \dots, n_v-1.$$

Prototype `int N_VScaleVectorArray_PSBLAS(int nvec, realtype* c, N_Vector* X, N_Vector* Z)`

`N_VConstVectorArray_PSBLAS` : This routine sets each element in a vector of n elements in a vector array of n_v vectors to the same value.

Prototype `int N_VConstVectorArray_PSBLAS(int nvecs, realtype c, N_Vector* Z)`

`N_VWrmsNormVectorArray_PSBLAS` : This routine computes the weighted root mean square norm of n_v vectors with n elements.

Prototype `int N_VWrmsNormVectorArray_PSBLAS(int nvecs, N_Vector* X, N_Vector* W, realtype* nrm)`

`N_VWrmsNormMaskVectorArray_PSBLAS` : This routine computes the masked weighted root mean square norm of n_v vectors with n elements.

Prototype `int N_VWrmsNormMaskVectorArray_PSBLAS(int nvec, N_Vector* X, N_Vector* W, N_Vector id, realtype* nrm)`

`N_VScaleAddMultiVectorArray_PSBLAS` : This routine scales and adds a vector in a vector array of n_v vectors to the corresponding vector in n_s vector arrays.

Prototype `int N_VScaleAddMultiVectorArray_PSBLAS(int nvec, int nsum, realtype* a, N_Vector* X, N_Vector** Y, N_Vector** Z)`

`N_VLinearCombinationVectorArray_PSBLAS` : This routine computes the linear combination of n_s vector arrays containing n_v vectors with n elements.

Prototype `int N_VLinearCombinationVectorArray_PSBLAS(int nvec, int nsum, realtype* c, N_Vector** X, N_Vector* Z)`

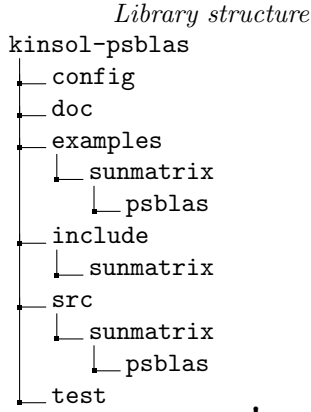
By default all fused and vector array operations are disabled in the `nvec` PSBLAS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector.

```

int N_VEnableFusedOps_PSBLAS(N_Vector v, boolean_t tf);
int N_VEnableLinearCombination_PSBLAS(N_Vector v, boolean_t
tf);
int N_VEnableScaleAddMulti_PSBLAS(N_Vector v, boolean_t tf);
int N_VEnableDotProdMulti_PSBLAS(N_Vector v, boolean_t tf);
int N_VEnableLinearSumVectorArray_PSBLAS(N_Vector v,
boolean_t tf);
int N_VEnableScaleVectorArray_PSBLAS(N_Vector v, boolean_t
tf);
int N_VEnableConstVectorArray_PSBLAS(N_Vector v, boolean_t
tf);
int N_VEnableWrmsNormVectorArray_PSBLAS(N_Vector v,
boolean_t tf);
int N_VEnableWrmsNormMaskVectorArray_PSBLAS(N_Vector v,
boolean_t tf);
int N_VEnableScaleAddMultiVectorArray_PSBLAS(N_Vector v,
boolean_t tf);
int N_VEnableLinearCombinationVectorArray_PSBLAS(N_Vector v,
boolean_t tf);

```

3 The SUNMATRIX_PSBLAS implementation



! →

The SUNMATRIX_PSBLAS implementation of the SUNDIALS SUNMATRIX module provides an interface to the PSBLAS code for handling distributed sparse matrices. It defines the *content* field of SUNMATRIX to be a structure containing the PSBLAS descriptor for the data distribution, a PSBLAS sparse matrix object of double, and the PSBLAS communicator (context).

```

struct _SUNMatrixContent_PSBLAS {
    psb_c_descriptor *cdh; /* descriptor for data distribution */
    psb_c_dspmat *ah; /* PSBLAS sparse matrix */
    int ictxt; /* PSBLAS communicator */
};

```

All the matrices that have to interact needs to be instantiated on the same parallel context ictxt, and on the same data distribution cdh, and this holds also for the vectors for the mixed type operation.

The header file to include when using this module is `sunmatrix_psblas.h`. The installed module library to link to is `libsundials_sunmatrixpsblas.a` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

3.1 SUNMATRIX_PSBLAS accessor macros

The following macros are provided to access the content of a SUNMATRIX_PSBLAS matrix. The suffix `_P` in the names denotes the fact that the data are in distributed memory.

```

#define SMLCONTENT_P(A)      ( (SUNMatrixContent_PSBLAS) (A->
content) )
#define SMLDESCRIPTOR_P(A)  ( SMLCONTENT_P(A)->cdh )
#define SMLPMAT_P(A)        ( SMLCONTENT_P(A)->ah )
#define SMLICTXT_P(A)        ( SMLCONTENT_P(A)->ictxt )

```

SMLCONTENT_P(A)	this macro gives access to the contents of the PSBLAS matrix <code>SUNMATRIX</code> .
SMLDESCRIPTOR_P(A), SMLPMAT_P(A)	these macros give instead individual access to the parts of the content of a PSBLAS parallel <code>SUNMATRIX</code> .
SMLICTXT_P(A)	this macro provides the PSBLAS context used by the SUNMATRIX_PSBLAS sparse matrices.

In PSBLAS every sparse matrix has an associated state, which can take one of the following values:

- BUILD : is the state entered after the first allocation, and before the first assembly; in this state it is possible to add nonzero entries.
- ASSEMBLED : is the state entered after the assembly; computations using the sparse matrix, such as matrix-vector products, are only possible in this state;
- UPDATE : state entered after a reinitialization; this is used to handle applications in which the same sparsity pattern is used multiple times with different coefficients. In this state *it is only possible to enter coefficients for already existing nonzero entries.*
- ! →

3.2 SUNMATRIX_PSBLAS functions

The SUNMATRIX_PSBLAS implementation provides PSBLAS implementations of all the sparse matrix operations listed in Table 7.2 of the original KINSOL library [1]. Following the standard nomenclature of the SUNDIALS library, their names are obtained from the ones listed in Table 7.2 by appending the suffix `_PSBLAS`. The SUNMATRIX_PSBLAS implementation provides the following additional user-callable routines

SUNMatAsb_PSBLAS : This routine assemble the SUNMATRIX after that all the elements have been inserted into it, i.e., after that all the calls to the **SUNMatIns_PSBLAS** routine have been completed. This is substantially a wrapper for the PSBLAS function `psb_c_dspasb`.

Prototype `int SUNMatAsb_PSBLAS(SUNMatrix A);`

SUNMatIns_PSBLAS : This routine inserts a set of coefficients into a sparse matrix. On entry to this routine the descriptor may be in either the BUILD or ASSEMBLED state, while the sparse matrix may be in either the BUILD or UPDATE state. We stress that it mandatory that if the descriptor is in the BUILD state, then also the sparse matrix *must* be in the BUILD state, since adding entries to the matrix causes internal calls altering the structure of the communication pattern for the distributed object. The insert of the element in the matrix is assumed to be in the COO format, thus the coefficients to be inserted are represented by the ordered triples `irw[i]`, `icl[i]`, `val[i]`, for $i = 1, \dots, nz$; these triples should belong to the current process, i.e., the index `irw[i]` should be one of the local indices, but are otherwise arbitrary, if this is not the case any coefficients from matrix rows not owned by the calling process are silently ignored.

Prototype `int SUNMatIns_PSBLAS(psb_i_t nz, const psb_l_t *irw, const psb_l_t *icl, const psb_d_t *val, SUNMatrix A);`

3.2.1 Description of the SUNMATRIX_PSBLAS functions

SUNPSBLASMatrix : Define and implement user-callable constructor routines to create a SUNMatrix on the PSBLAS context `ictxt` with the communicator `cdh`, and with the content field and ops pointing to the matrix operations defined in the following. The matrix is initialized in the BUILD state.

Prototype `SUNMatrix SUNPSBLASMatrix(int ictxt, psb_c_descriptor *cdh)`

SUNPSBLASMatrix_Print : This function prints the content of a dense SUNMatrix in Matrix-Market format to file specified by `filename`, the routine takes care of creating/opening the file, a separated one for each process in which the local part (without any eventual overlap) is printed.

Prototype	<code>void SUNPSBLASMatrix_Print(SUNMatrix A, char *matrixtitle, char *filename)</code>
<code>SUNPSBLASMatrix_Rows</code> :	This function returns the number of (global) rows in the PSBLAS SUNMatrix.
Prototype	<code>sunindextype SUNPSBLASMatrix_Rows(SUNMatrix A)</code>
<code>SUNPSBLASMatrix_Columns</code> :	This function returns the number of (global) columns in the PSBLAS SUNMatrix.
Prototype	<code>sunindextype SUNPSBLASMatrix_Columns(SUNMatrix A)</code>
<code>SUNPSBLASMatrix_NNZ</code> :	This function returns the (global) number of non-zero entries in the PSBLAS SUNMatrix.
Prototype	<code>sunindextype SUNPSBLASMatrix_NNZ(SUNMatrix A)</code>
<code>SUNMatGetID_PSBLAS</code> :	Returns the type identifier for the matrix A, since this is a custom implementation it returns the value <code>SUNMATRIX_CUSTOM</code> .
Prototype	<code>SUNMatrix_ID SUNMatGetID_PSBLAS(SUNMatrix A)</code>
<code>SUNMatClone_PSBLAS</code> :	Creates a new SUNMatrix on the same descriptor of an existing matrix A and sets the <code>ops</code> field. It does not copy the matrix, but rather allocates storage for the new matrix leaving the new matrix in the BUILD state.
Prototype	<code>SUNMatrix SUNMatClone_PSBLAS(SUNMatrix A)</code>
<code>SUNMatDestroy_PSBLAS</code> :	Destroys the SUNMatrix A and frees memory allocated for its internal data. This routine does not free the communicator, in general there are many objects insisting on the same communicator, therefore it should be destroyed/freed on its own.
Prototype	<code>void SUNMatDestroy_PSBLAS(SUNMatrix A)</code>
<code>SUNMatZero_PSBLAS</code> :	Performs the operation $(A)_{i,j} = 0$ for all entries of the matrix A. The return value is an integer flag denoting success/failure of the operation. The PSBLAS at the end of this operation is left in the UPDATE state.
Prototype	<code>int SUNMatZero_PSBLAS(SUNMatrix A)</code>
<code>SUNMatCopy_PSBLAS</code> :	Performs the operation $(B)_{i,j} = (A)_{i,j}$ for all entries of the matrices A and B. The return value is an integer flag denoting success/failure of the operation. The matrix B inherits the state of the matrix A.
Prototype	<code>int SUNMatCopy_PSBLAS(SUNMatrix A, SUNMatrix B)</code>
<code>SUNMatScaleAdd_PSBLAS</code> :	Performs the operation $A = cA + B$. The return value is an integer flag denoting success/failure of the operation. This function assumes that both the matrices are defined on the same communicator.
Prototype	<code>int SUNMatScaleAdd_PSBLAS(realtype c, SUNMatrix A, SUNMatrix B)</code>
<code>SUNMatScaleAddI_PSBLAS</code> :	Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.
Prototype	<code>int SUNMatScaleAddI_PSBLAS(realtype c, SUNMatrix A)</code>
<code>SUNMatMatvec_PSBLAS</code> :	Performs the matrix-vector product operation, $y = Ax$. It should only be called with vectors x and y that are compatible with the

matrix A , i.e., they should be both defined on the same communicator and have the same dimensions. The return value is an integer flag denoting success/failure of the operation.

Prototype `int SUNMatMatvec_PSBLAS(SUNMatrix A, N_Vector x, N_Vector y)`

`SUNMatSpace_PSBLAS` : This function is advisory only, for use in determining a user's total space requirements, for this module it is a dummy function always returning true.

Prototype `int SUNMatSpace_PSBLAS(SUNMatrix A, long int *lenrw, long int *leniw)`

3.2.2 An example of matrix assembly

As an example of usage of this matrix routines we consider the same boundary value problem in (2), to build the matrix A associated to the centered finite difference discretization

$$\dots \quad (3)$$

To this end we use the same communicator `cdh` and context `ictxt` we have allocated in Listing 1, by simply doing:

```
A = SUNPSBLASMatrix(ictxt, cdh);
matgen(ictxt, nl, idim, vl, A);
psb_c_cdasb(cdh);
SUNMatAsb_PSBLAS(A);
```

We have first initialized the sparse PSBLAS matrix inside the SUNMATRIX container, then by using the `matgen` allocation routine in Listing 3 we have populated the sparse matrix, and finally assembled both the descriptor, and the sparse matrix.

The complete example can be found in `examples/sunmatrix/psblas`.

4 The SUNLINSOL_PSBLAS implementation

Library structure

```
kinsol-psblas
├── config
├── doc
├── examples
│   └── sunlinsol
│       └── psblas
├── include
│   └── sunlinsol
├── src
│   └── sunlinsol
│       └── psblas
└── test
```

The SUNLINSOL_PSBLAS implementation of the SUNDIALS SUNLINSOL module provides an interface to the PSBLAS code for handling the iterative solution of large and sparse linear systems. It defines the `content` field of `SUNLinsol` to be a structure containing the options needed to setup a PSBLAS solver, the PSBLAS descriptor for the data distribution, the pointer to both PSBLAS and MLD2P4 preconditioner, the PSBLAS sparse matrix for which the solver (and the preconditioner) is instantiated, the PSBLAS communicator (context), and two strings that identify what iterative method and type of preconditioner we are dealing with.

```
struct _SUNLinearSolverContent_PSBLAS {
    psb_c_SolverOptions options; /* PSBLAS solver options */
    psb_c_dprec *ph;             /* PSBLAS preconditioner */
    mld_c_dprec *mh;             /* MLD2P4 preconditioner */
    psb_c_descriptor *cdh;        /* Descriptor */
    psb_c_dspmat *ah;             /* PSBLAS sparse matrix */
    int ictxt;                   /* PSBLAS communicator */
    char methd[40];              /* String for Method */
    char ptype[20];              /* String for Preconditioner */
};
```

The structure `options` contains the options that are common in between the different iterative solver included in PSBLAS that are interfaced with this library

```

typedef struct psb_c_solveroptions {
    int iter; /* On exit how many iterations were performed */
    int itmax; /* On entry maximum number of iterations */
    int itrace; /* On entry print an info message every itrace
                iterations */
    int irst; /* Restart depth for RGMRES or BiCGSTAB(L) */
    int istop; /* Stopping criterion: 1:backward error 2: ||r||
                _2/||b||_2 */
    double eps; /* Stopping tolerance */
    double err; /* Convergence indicator on exit */
} psb_c_SolverOptions;

```

The default settings for this structure can be set by means of the `psb_c_DefaultSolverOptions` routine

Prototype `int psb_c_DefaultSolverOptions(psb_c_SolverOptions *options)` that sets the following default values

```

options.itmax = 1000;
options.itrace = 0;
options.istop = 2;
options.irst = 10;
options.eps = 1.d-6;
err = 0;

```

Specifically, the Krylov method included from the PSBLAS library are given in Table 1, and that permit to solve a wide range of problems.

methd	Method	Type of Matrix
"CG"	Conjugate Gradient	SPD
"CGS"	Conjugate Gradient Stabilized	General
"GCR"	Generalized Conjugate Residual	SPD
"FCG"	Flexible Conjugate Gradient	SPD
"BICG"	Bi-Conjugate Gradient	General
"BICGSTAB"	Bi-Conjugate Gradient Stabilized	General
"BICGSTABL"	Restarted Bi-Conjugate Gradient Stabilized	General
"RGMRES"	Restarted Generalized Minimal Residual	General

Table 1: Krylov methods included with the PSBLAS library.

The general type of preconditioner can be instead selected by means of the string `ptype`, that can assume three values for the basic preconditioner implemented in the core PSBLAS library that are

- "NONE" : no preconditioning is used, i.e., the preconditioner is just a copy operator acting on the residuals,
- "BJAC" : precondition by means of a factorization of the block-diagonal of matrix A, the block boundaries are determined by using the data allocation boundaries that have been defined for each process. It is implemented to require no communication, to solve the system on the blocks the incomplete ILU(0) factorization is used,
- "DIAG" : this is a simple diagonal scaling; each entry of the input vector is multiplied by the reciprocal of the sum of the absolute values of the coefficients in the corresponding row of matrix A.

Each of this choices select the routines that work with the `psb_c_dprec *ph` in the SUNLINSOL content and leaves `mld_c_dprec *mh` a NULL pointer.

The more elaborated *multilevel* preconditioners are enable by setting `ptype` to one of the following string

- "GS" : hybrid Gauss-Seidel (forward), that is, a global block Jacobi preconditioner with Gauss-Seidel as local solver on the blocks,
- "FBGS" : symmetrized hybrid Gauss-Seidel, that is, forward Gauss-Seidel followed by backward Gauss-Seidel.
- "AS" : Additive Schwarz (AS), with overlap 1 and ILU(0) on the local blocks.
- "ML" : this enables the construction of a multilevel preconditioner for which all the details for these preconditioner are described in Section 4.3, and are based on the ML2DP4 package [2, 4].

This choice selects the routines that work with the `mld_c_dprec *mh` in the `SUNLINSOL` content and leaves `psb_c_dprec *ph` a `NULL` pointer.

4.1 SUNLINSOL_PSBLAS accessor macros

The following macros are provided to access the content of a `SUNLINSOL_PSBLAS` linear solver. The suffix `_P` in the names denotes the fact that the data are in distributed memory.

```
#define PSBLAS_CONTENT(S) ( (SUNLinearSolverContent_PSBLAS)(
    S->content) )
#define LS_PREC_P(S) ( PSBLAS_CONTENT(S)->ph )
#define LS_MLPREC_P(S) ( PSBLAS_CONTENT(S)->mh )
#define LS_DESCRIPTOR_P(S) ( PSBLAS_CONTENT(S)->cdh )
#define LS_PMAT_P(S) ( PSBLAS_CONTENT(S)->ah )
#define LS_IXTXT_P(S) ( PSBLAS_CONTENT(S)->ixtxt )
#define LS_METHD_P(S) ( PSBLAS_CONTENT(S)->methd )
#define LS_PTYPE_P(S) ( PSBLAS_CONTENT(S)->ptype )
```

<code>PSBLAS_CONTENT(S)</code>	this macro gives access to the contents of the PSBLAS linear solver <code>SUNLINSOL</code> .
<code>LS_PREC_P(S)</code>	this macro gives access to the pointer to the PSBLAS preconditioner object.
<code>LS_MLPREC_P(S)</code>	this macro gives access to the pointer to the MLD2P4 preconditioner object.
<code>LS_DESCRIPTOR_P(S)</code> , <code>LS_PMAT_P(S)</code>	these macros give instead individual access to the parts of the content of a PSBLAS parallel <code>SUNMATRIX</code> .
<code>LS_IXTXT_P(S)</code>	this macro provides the PSBLAS context used by the <code>SUNMATRIX_PSBLAS</code> sparse matrices and vectors.
<code>LS_METHD_P(S)</code>	this macro gives access to the string that select the type of Krylov solver to be used as iterative method.
<code>LS_PTYPE_P(S)</code>	this macro gives access to the string that select the type of preconditioner to be used within the iterative method.

4.2 SUNLINSOL_PSBLAS functions

The `SUNLINSOL_PSBLAS` implementation provides PSBLAS implementations of all the linear solver operations listed in `KINSOL` library [1, Section 8.1.1]. Following the standard nomenclature of the `SUNDIALS` library, their names are obtained from the ones listed there by appending the suffix `_PSBLAS`. The `SUNLINSOL_PSBLAS` implementation provides the following additional user-callable routines that can be used to set-up the values for the `ptype="ML"` preconditioners.

SUNLinSolSeti_PSBLAS : given a linear solver with `ptype="ML"` this routine set the option `const char *what` to the given integer value

Prototype `int SUNLinSolSeti_PSBLAS(SUNLinearSolver S, const char *what, psb_int val)`

SUNLinSolSetc_PSBLAS : given a linear solver with `ptype="ML"` this routine set the option `const char *what` to the given string value

Prototype `int SUNLinSolSetc_PSBLAS(SUNLinearSolver S, const char *what, const char *val)`

SUNLinSolSetr_PSBLAS : given a linear solver with `ptype="ML"` this routine set the option `const char *what` to the given `double` val

Prototype `int SUNLinSolSetr_PSBLAS(SUNLinearSolver S, const char *what, double val)`

The detail of the options for the various solver are recalled in Section 4.3, and are fully detailed in [4].

4.2.1 Description of the SUNLINSOL_PSBLAS functions

SUNLinSol_PSBLAS : Only the solver options are set at this stage, all the information regarding the communicator, the PSBLAS contex are imported from the matrix when the solver is initialized. The setup of the preconditioner has to be done with the `SUNLinSolSetup_PSBLAS` routine. Here the codes decide only if we are using a PSBLAS or an MLD2P4 preconditioner by looking at the string `char ptype[]`.

Prototype `SUNLinearSolver SUNLinSol_PSBLAS(psb_c.SolverOptions options, char method[], char ptype[])`

SUNLinSolGetType_PSBLAS : this function returns the type identifier for the linear solver, that in this case is of matrix-iterative type.

Prototype `SUNLinearSolver_Type SUNLinSolGetType_PSBLAS(SUNLinearSolver S)`

SUNLinSolInitialize_PSBLAS : this function performs the linear solver initialization by assuming that all solver-specific options have been set, i.e., in the case of `ptype="ML"` preconditioner after the calls to the `SUNLinSolSet[i,c,r]_PSBLAS` routines.

Prototype `int SUNLinSolInitialize_PSBLAS(SUNLinearSolver S)`

SUNLinSolSetup_PSBLAS : this function performs the linear solver setup, based on an (possibly) updated system `SUNMATRIX A`. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.

Prototype `int SUNLinSolSetup_PSBLAS(SUNLinearSolver S, SUNMatrix A)`

SUNLinSolSolve_PSBLAS : this function performs the actual solve of the linear system $A\mathbf{x} = \mathbf{b}$ (by eventually using the preconditioner P^{-1} encoded in the `SUNLinearSolver S`).

Prototype `int SUNLinSolSolve_PSBLAS(SUNLinearSolver S, SUNMatrix A, N_Vector x, N_Vector b, realtype tol)`

SUNLinSolFree_PSBLAS : this function frees memory allocated by the linear solver, it is important to notice that this routine frees only the preconditioner and

the structure containing the various part of the solver: the communicator and the matrix are still there, they should be freed after the matrix has been destroyed.

Prototype `int SUNLinSolFree_PSBLAS(SUNLinearSolver S)`

`SUNLinSolNumIters_PSBLAS` : return the number of linear iterations performed in the last solve call.

Prototype `int SUNLinSolNumIters_PSBLAS(SUNLinearSolver S)`

`SUNLinSolResNorm_PSBLAS` : return the residual of the last linear iteration performed in the last solve call.

Prototype `realtype SUNLinSolResNorm_PSBLAS(SUNLinearSolver S)`

`SUNLinSolLastFlag_PSBLAS` : return the last error flag encountered within the linear solver. This is not called by the sundials packages directly; it allows the user to investigate linear solver issues after a failed solve.

Prototype `long int SUNLinSolLastFlag_PSBLAS(SUNLinearSolver S)`

4.3 Algebraic Multigrid Preconditioners: the MLD2P4 package

We consider here the classic set of an algebraic multigrid preconditioner. Thus we have as finest index space the set of row (column) indices of A , i.e., $\Omega = \{1, 2, \dots, n\}$. Any of the algebraic multilevel preconditioners that is implemented in the MLD2P4 package generates a hierarchy of index spaces and a corresponding hierarchy of matrices,

$$\Omega^1 \equiv \Omega \supset \Omega^2 \supset \dots \supset \Omega^{nlev}, \quad A^1 \equiv A, A^2, \dots, A^{nlev},$$

by using the information contained in A , **without assuming any knowledge of the geometry of the problem** from which A originates. A vector space \mathbb{R}^{n_k} is associated with Ω^k , where n_k is the size of Ω^k . For all $k < nlev$, a restriction operator and a prolongation one are built, which connect two levels k and $k + 1$:

$$P^k \in \mathbb{R}^{n_k \times n_{k+1}}, \quad R^k \in \mathbb{R}^{n_{k+1} \times n_k},$$

the matrix A^{k+1} is computed by using the previous operators according to the Galerkin approach, i.e.,

$$A^{k+1} = R^k A^k P^k.$$

For the construction of R^k we have that $R^k = (P^k)^T$. A smoother with iteration matrix M^k is set up at each level $k < nlev$, and a solver is set up at the coarsest level, so that they are ready for application.

All this construction is what is usually called the build-phase of the hierarchy of AMG components, i.e., the so-called build phase of the preconditioner.

The MLD2P4 package offers a plethora of way to obtain all the ingredient needed to build what is described here. These are explained in Tables 2 to 8 in [4] and, depending on the type of option, can all be set by means of the `SUNLinSolSet[i/c/r]_PSBLAS` routines.

We refer back to the guide [4] for the full details on how to setup the preconditioner.

4.3.1 An example of linear system solution

We can consider the same context of Example 1, with the matrix construction detail in Section 3.2.2. Now that both the data distribution, and the construction of the (distributed) sparse matrix A and right hand-side \mathbf{b} have been completed we only need to create the SUNLINSOL object, and setup the preconditioner.

```
SUNLinearSolver LS;                                /* linear solver object */
psb_c_SolverOptions options;                        /* Solver options */
/* Set up the solver options */
psb_c_DefaultSolverOptions(&options);
options.eps = tol;
options.itmax = itmax;
options.irst = irst;
options.itrace = 1;
options.istop = istop;
/* Create PSBLAS/MLD2P4 linear solver */
LS = SUNLinSol_PSBLAS(options, "CG", "ML");
SUNLinSolInitialize(LS);
SUNLinSolSeti_PSBLAS(LS, "SMOOTHER_SWEEPS", 2);
SUNLinSolSeti_PSBLAS(LS, "SUB_FILLIN", 1);
SUNLinSolSetc_PSBLAS(LS, "COARSE_SOLVE", "BJAC");
SUNLinSolSetc_PSBLAS(LS, "COARSE_SUBSOLVE", "ILU");
SUNLinSolSeti_PSBLAS(LS, "COARSE_FILLIN", 0);
SUNLinSolSetup(LS, A);
SUNLinSolSolve_PSBLAS(LS, A, x, b, tol);
```

Listing 2: "An example of solution with PCG preconditioned by a ML preconditioner"

The example given in the code 2, initializes a linear solver requesting for a Conjugate Gradient method ("CG"), preconditioned by a multi-level preconditioner ("ML"). Then, after the initialization step, some of the properties of the multigrid cycle are selected. Specifically, we decide to use two smoother sweeps of Block-Jacobi with an ILU(1) subsolver ("SUB_FILLIN"), and set again a Block-Jacobi method as coarse solver ("COARSE_SUBSOLVE") that uses instead as solver for the blocks and ILU(0) ("COARSE_FILLIN") factorization. Then a setup step is performed, during this step the multigrid hierarchy is assembled. Finally the solution of the linear system is computed by a call to SUNLinSolSolve_PSBLAS. Observe that if we have put the value `options.itrace = 1`; the solve routine will print convergence information for each iteration.

A Matrix assembly routine

The following routine loops through the local entries of the communicator allocate in Listing 1, and uses the `SUNMatIns_PSBLAS` routine to insert the entries in the sparse matrix. The auxiliary functions $\{a_i, b_i, g\}_{i=1}^3$ take care of the coefficient functions in (2).

```
psb_i_t matgen(psb_i_t ictxt, psb_i_t nl, psb_i_t idim,
              psb_l_t vl[], SUNMatrix A)
{
    psb_i_t iam, np;
    psb_l_t ix, iy, iz, el, glob_row;
    psb_i_t i, k, info;
    double x, y, z, deltah, sqdeltah, deltah2;
    double val[10*NBMAX], zt[NBMAX];
    psb_l_t irow[10*NBMAX], icol[10*NBMAX];

    info = 0;
    psb_c_info(ictxt, &iam, &np);
    deltah = (double) 1.0/(idim+1);
```

```

sqdeltah = deltah*deltah;
deltah2 = 2.0* deltah;
psb_c_set_index_base(0);
for (i=0; i<nl; i++) {
glob_row=v1[i];
el=0;
ix = glob_row/(idim*idim);
iy = (glob_row-ix*idim*idim)/idim;
iz = glob_row-ix*idim*idim-iy*idim;
x=(ix+1)*deltah;
y=(iy+1)*deltah;
z=(iz+1)*deltah;
zt[0] = 0.0;
/* internal point: build discretization */
/* term depending on (x-1,y,z) */
val[el] = -a1(x,y,z)/sqdeltah-b1(x,y,z)/deltah2;
if (ix==0) {
zt[0] += g(0.0,y,z)*(-val[el]);
} else {
icol[el]=(ix-1)*idim*idim+(iy)*idim+(iz);
el=el+1;
}
/* term depending on (x,y-1,z) */
val[el] = -a2(x,y,z)/sqdeltah-b2(x,y,z)/deltah2;
if (iy==0) {
zt[0] += g(x,0.0,z)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy-1)*idim+(iz);
el=el+1;
}
/* term depending on (x,y,z-1) */
val[el]=-a3(x,y,z)/sqdeltah-b3(x,y,z)/deltah2;
if (iz==0) {
zt[0] += g(x,y,0.0)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy)*idim+(iz-1);
el=el+1;
}
/* term depending on (x,y,z) */
val[el]=2.0*(a1(x,y,z)+a2(x,y,z)+a3(x,y,z))/sqdeltah + c(x,y,
z);
icol[el]=(ix)*idim*idim+(iy)*idim+(iz);
el=el+1;
/* term depending on (x,y,z+1) */
val[el] = -a3(x,y,z)/sqdeltah+b3(x,y,z)/deltah2;
if (iz==idim-1) {
zt[0] += g(x,y,1.0)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy)*idim+(iz+1);
el=el+1;
}
/* term depending on (x,y+1,z) */
val[el] = -a2(x,y,z)/sqdeltah+b2(x,y,z)/deltah2;
if (iy==idim-1) {
zt[0] += g(x,1.0,z)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy+1)*idim+(iz);
el=el+1;
}
/* term depending on (x+1,y,z) */
val[el] = -a1(x,y,z)/sqdeltah+b1(x,y,z)/deltah2;
if (ix==idim-1) {
zt[0] += g(1.0,y,z)*(-val[el]);
} else {
icol[el]=(ix+1)*idim*idim+(iy)*idim+(iz);
el=el+1;
}
}
for (k=0; k<el; k++) irow[k]=glob_row;
if ((info=SUNMatIns_PSBLAS(el,irow,icol,val,A))!=0)
fprintf(stderr,"From psb_c_dspins: %d\n",info);

```

```

}

return (info);
}

```

Listing 3: "Allocation routine for the discrete boundary value problem (2)."

Observe that to speed-up the insertion procedure we are collecting together a certain number of rows to be inserted, specifically $10 \cdot \text{NBMAX}$ for a defined value of NBMAX. It is clearly possible to execute one call for each nonzero coefficient, however this would have a substantial computational overhead. Therefore packing a "certain amount of data" ($10 \cdot \text{NBMAX}$) into each call to the insertion routine is advisable. The best performing value of NBMAX depends on both the architecture of the computer being used and on the problem structure.

References

- [1] Aaron M. Collier et al. *User Documentation for kinsol v4.1.0*. Center for Applied Scientific Computing Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/sites/default/files/public/kin_guide.pdf.
- [2] Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. "MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95". In: *ACM Trans. Math. Software* 37.3 (2010), Art. 30, 23. ISSN: 0098-3500. DOI: 10.1145/1824801.1824808. URL: <https://doi.org/10.1145/1824801.1824808>.
- [3] S. Filippone and A. Buttari. *PSBLAS 3.6.1 User's guide. A reference guide for the Parallel Sparse BLAS library*. Cranfield University, Centre for Computational Engineering Sciences. URL: <https://github.com/sfilippone/psblas3/blob/development/docs/psblas-3.6.pdf>.
- [4] D. Di Serafino P. D'Ambra and S. Filippone. *MLD2P4 User's and Reference Guide. A guide for the MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS*. Cranfield University, Centre for Computational Engineering Sciences. URL: <https://github.com/sfilippone/mld2p4-2/blob/development/docs/mld2p4-2.2-guide.pdf>.