
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 9, 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	4
3	The SUNMATRIX_PSBLAS implementation	9
4	The SUNLINSOL_PSBLAS implementation	9
4.1	Algebraic Multigrid Preconditioners	9

1 The logic behind a PSBLAS instrumented application

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

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

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 .

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.

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

$$-\frac{b_1 \partial^2 u}{\partial x^2} - \frac{b_2 \partial^2 u}{\partial y^2} + \frac{a_1 \partial u}{\partial x} + \frac{a_2 \partial u}{\partial y} + a_3 u = 0, \quad (x, y, z) \in [0, 1]^3,$$

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;
```

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. 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 [2].

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_psblas.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_ICTXT_P(v) (NV_CONTENT_P(v)->ictxt)
```

<code>NV_CONTENT_P(v)</code>	this macro gives access to the contents of the PSBLAS vector <code>N_Vector</code> .
<code>NV_DESCRIPTOR_P(v),</code> <code>NV_OWNDATA_P(v),</code> <code>NV_PVEC_P(v)</code>	these macros give instead individual access to the parts of the content of a PSBLAS parallel <code>N_Vector</code> .
<code>NV_ICTXT_P(v)</code>	this macro provides the PSBLAS context used by the NVECTOR_PSBLAS 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 routines.

<code>N_VNew_PSBLAS</code>	: This function creates and allocates memory for a parallel vector on the PSBLAS context <code>ictxt</code> with the communicator <code>cdh</code>
----------------------------	--

Prototype	<code>N_Vector N_VNew_PSBLAS(int ictxt, psb_c_descriptor *cdh);</code>
-----------	--

<code>N_VNewEmpty_PSBLAS</code>	: This function creates a new PSBLAS vector with empty data array.
---------------------------------	--

Prototype	<code>N_Vector N_VNewEmpty_PSBLAS(int ictxt, psb_c_descriptor *cdh);</code>
-----------	---

<code>N_VMake_PSBLAS</code>	:	Function to create a PSBLAS <code>N_Vector</code> with user data component. This function is substantially a wrapper for the PSBLAS function <code>psb_c_dgeins</code> .
Prototype		<code>N_Vector N_VMake_PSBLAS(int ictxt, psb_c_descriptor *cdh, psb_i_t m, psb_i_t *irow, double *val);</code>
		The PSBLAS context <code>ictxt</code> with the communicator <code>cdh</code> are the one defined for the whole programs, the integer <code>m</code> is the number of rows in <code>val[]</code> to be inserted, the array of integers <code>irow</code> is the indices of the rows to be inserted. Specifically, row <code>i</code> of <code>val</code> will be inserted into the local row corresponding to the global index <code>row index row[i]</code> .
! →		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 <code>N_VAsb_PSBLAS</code> routine.
<code>N_VAsb_PSBLAS</code>	:	This routine assemble the <code>N_Vector</code> after that all the elements have been inserted into it, i.e., after that all the calls to the <code>N_VMake_PSBLAS</code> routine have been completed. This is substantially a wrapper for the PSBLAS function <code>psb_c_dgeasb</code> .
Prototype		<code>void N_VAsb_PSBLAS(N_Vector v)</code>
<code>N_VCloneVectorArray_PSBLAS</code>	:	This function creates an array of new parallel vectors (by cloning) an array of count parallel vectors <code>v</code> .
Prototype		<code>N_Vector *N_VCloneVectorArray_PSBLAS(int count, N_Vector w)</code>
<code>N_VCloneVectorArrayEmpty_PSBLAS</code>	:	This function creates an array of count new parallel vectors with empty data array on the same communicator and context of the vector <code>w</code> .
Prototype		<code>N_Vector *N_VCloneVectorArrayEmpty_PSBLAS(int count, N_Vector w)</code>
<code>N_VDestroyVectorArray_PSBLAS</code>	:	This function to frees an array of count <code>N_Vectors</code> created with <code>N_VCloneVectorArray_PSBLAS</code>
Prototype		<code>void N_VDestroyVectorArray_PSBLAS(N_Vector *vs, int count)</code>
<code>N_VGetLength_PSBLAS</code>	:	This function returns the <i>global</i> vector length, this is substantially a wrapper for the PSBLAS function <code>psb_c_cd_get_global_rows</code> .
Prototype		<code>sunindextype N_VGetLength_PSBLAS(N_Vector v)</code>
<code>N_VGetLocalLength_PSBLAS</code>	:	This function returns the <i>local</i> vector length, this is substantially a wrapper for the PSBLAS function <code>psb_c_cd_get_local_rows</code> .
Prototype		<code>sunindextype N_VGetLocalLength_PSBLAS(N_Vector v)</code>
<code>N_VPrint_PSBLAS</code>	:	This function prints the local data in a parallel vector to stdout.
Prototype		<code>void N_VPrint_PSBLAS(N_Vector x)</code>
<code>N_VPrintFile_PSBLAS</code>	:	This function prints the local data in a parallel vector to <code>outfile</code> .
Prototype		<code>void N_VPrintFile_PSBLAS(N_Vector x, FILE* outfile)</code>
<code>N_VGetVectorID_PSBLAS</code>	:	This function returns the SUNDIALS identificative for the PSBLAS vector, since this is a custom implementation it returns the integer constant <code>SUNDIALS_NVEC_CUSTOM</code> .
Prototype		<code>N_Vector_ID N_VGetVectorID_PSBLAS(N_Vector v)</code>
<code>N_VCloneEmpty_PSBLAS</code>	:	Clones a <code>NVECTOR_PSBLAS</code> with a <code>NULL</code> <code>pvec</code> field, and with value <code>SUNFALSE</code> in the <code>own_data</code> field.

Prototype	<code>N_Vector N_VCloneEmpty_PSBLAS(N_Vector w)</code>
<code>N_VClone_PSBLAS</code> :	Clones a NVECTOR_PSBLAS allocating its memory following the same communicator of the cloned one.
Prototype	<code>N_Vector N_VClone_PSBLAS(N_Vector w)</code>
<code>N_VDestroy_PSBLAS</code> :	Destroys a NVECTOR_PSBLAS freeing both the memory allocated for the corresponding PSBLAS vector, and the memory allocated for the NVECTOR_PSBLAS structure.
Prototype	<code>void N_VDestroy_PSBLAS(N_Vector v)</code>
<code>N_VSpace_PSBLAS</code> :	Returns storage requirements for one NVECTOR_PSBLAS. <code>lrw</code> contains the number of realtype words and <code>liw</code> contains the number of integer words.
Prototype	<code>void N_VSpace_PSBLAS(N_Vector v, sunindextype *lrw, sunindextype *liw)</code>
<code>N_VGetArrayPointer_PSBLAS</code> :	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	<code>realtype *N_VGetArrayPointer_PSBLAS(N_Vector v)</code>
<code>N_VSetArrayPointer_PSBLAS</code> :	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 happens.
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 <code>c</code> , 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 happens.

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>
<code>N_VMin_PSBLAS</code> :	Gives back the minimum entry of an NVECTOR_PSBLAS object.
Prototype	<code>realtype N_VMin_PSBLAS(N_Vector x)</code>
<code>N_VWL2Norm_PSBLAS</code> :	Returns the weighted Euclidean 2-norm of the NVECTOR_PSBLAS x with realtype weight vector w.
Prototype	<code>realtype N_VWL2Norm_PSBLAS(N_Vector x, N_Vector w)</code>
<code>N_VL1Norm_PSBLAS</code> :	Computes the 1-norm of an NVECTOR_PSBLAS object.
Prototype	<code>realtype N_VL1Norm_PSBLAS(N_Vector x)</code>
<code>N_VCompare_PSBLAS</code> :	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	<code>void N_VCompare_PSBLAS(realtype c, N_Vector x, N_Vector z)</code>
<code>N_VInvTest_PSBLAS</code> :	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	<code>boolean N_VInvTest_PSBLAS(N_Vector x, N_Vector z)</code>
<code>N_VConstrMask_PSBLAS</code> :	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`type 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

`real`type N_VMinQuotient_PSBLAS(`N_Vector` num, `N_Vector` denom)

2.2.1 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, `real`type* 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 nv scalars.

Prototype

`int` N_VScaleAddMulti_PSBLAS(`int` nvec, `real`type* a, `N_Vector` x, `N_Vector`* Y, `N_Vector`* Z)

`N_VDotProdMulti_PSBLAS` :

This routine computes the dot product of a vector with nv other vectors.

Prototype

`int` N_VDotProdMulti_PSBLAS(`int` nvec, `N_Vector` x, `N_Vector`* Y, `real`type* dotprods)

2.2.2 Vector array operations

`N_VLinearSumVectorArray_PSBLAS` :

This routine computes the linear sum of two vector arrays containing nv 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, `real`type a, `N_Vector`* X, `real`type 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, `real`type* 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 nv vectors to the same value.

Prototype

`int` N_VConstVectorArray_PSBLAS(`int` nvecs, `real`type 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

4 The SUNLINSOL_PSBLAS implementation

4.1 Algebraic Multigrid Preconditioners

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] 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>.