PSBLAS-KINSOL interface*

Istituto per le Applicazioni del Calcolo "M. Picone",
Consiglio Nazionale delle Ricerche
Pasqua D'Ambra
Fabio Durastante
Salvatore Filippone
PSBLAS development — Kinsol Interface Version 1

May 14, 2020

Contents

* Work supported by the EC under the Horizon 2020 Project "Energy oriented Centre of Excellence for computing applications" (EoCoE II), Project ID: 824158

1 SUNDIALS KINSOL: Newton–Krylov Solvers

The original SUNDIALS library is a SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. It consists of the following six solvers:

- CVODE, solves initial value problems for ordinary differential equation (ODE) systems;
- CVODES, solves ODE systems and includes sensitivity analysis capabilities (forward and adjoint);
- ARKODE, solves initial value ODE problems with additive Runge-Kutta methods, include support for IMEX methods;
- IDA, solves initial value problems for differential-algebraic equation (DAE) systems;
- IDAS, solves DAE systems and includes sensitivity analysis capabilities (forward and adjoint);
- KINSOL, solves nonlinear algebraic systems.

The numerical operations performed in these codes are operations on vectors and matrices, and the codes provides both custom implementation and interfaces to these vector operations. What we provide here is a particular interface for such operations to the linear algebra operations provided by the Parallel Sparse BLAS (PSBLAS) library [5, 6]; see the whole structure of the library in Figure 1.

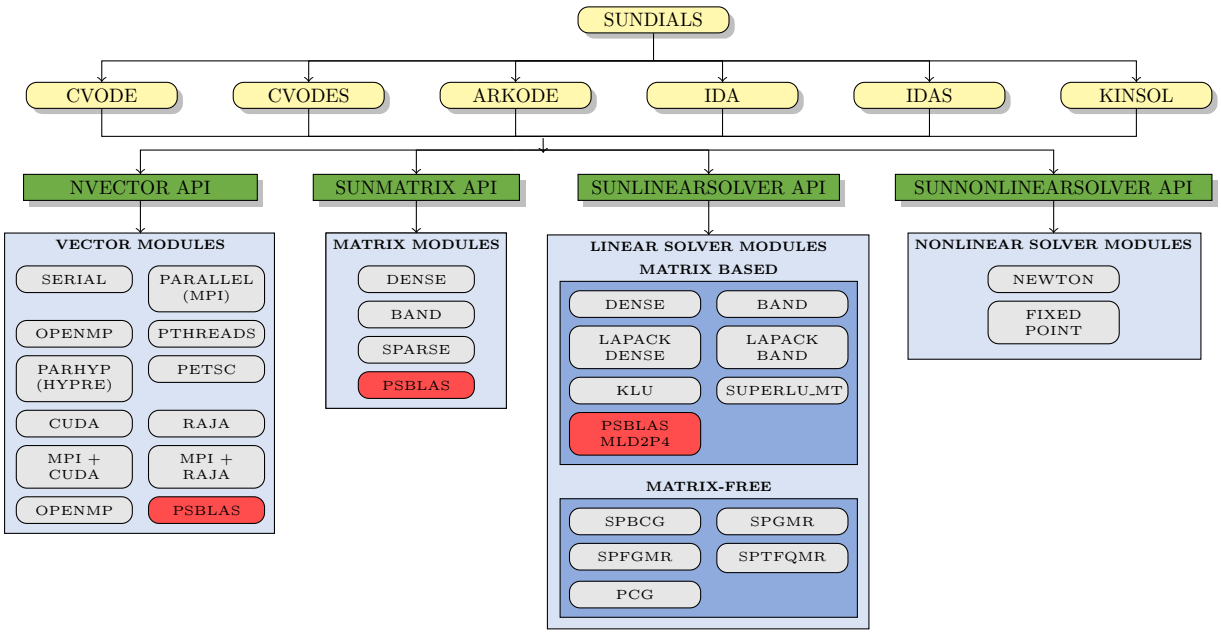


Figure 1: High-level diagram of the sundials suite, the additions presented here are highlighted in red.

Furthermore, many of the solvers included in SUNDIALS require the solution of linear systems of algebraic equations. For this task we provide a suite of parallel matrix–based Krylov solvers and preconditioners from the Multilevel Domain Decomposition Parallel Preconditioners Packages based on PSBLAS (MLD2P4) [3].

Our main objective is enabling the solution of nonlinear algebraic systems $F(\mathbf{x}) = 0$, for $\mathbf{x} \in \mathbf{R}^n$, by means of Newton–Krylov solvers. The rest of

this documentation presents the extension of the KINSOL package [1] NVECTOR, SUNMATRIX and SUNLINSOL API with the functionalities offered by PSBLAS/MLD2P4.

1.1 How to install

The first step for using these interfaces is to have a working version of both the PSBLAS and MLD2P4 library. The installation chain is therefore the following. As a first step you need to have all the auxiliary packages needed/suggested for the PSBLAS library, e.g., a local implementation of the BLAS library, the LAPACK library, METIS and AMD libraries. Then the development version from

`https://github.com/sfilippone/psblas3`

! → can be cloned, and installed; please refer to its manual for the complete details on the install procedure. If you want to compile KINSOL as a dynamic library it is necessary to pass the −fPic −ldl flags to the PSBLAS configure. The next packages that you need to install is the package of extensions for PSBLAS from

`https://github.com/sfilippone/psblas3-ext`.

The next step is then to install the suite of preconditioners, i.e., the MLD2P4 package. This packages has on its own a certain number of interfaces to other scientific libraries that are used to perform certain operation within it, e.g., the MUMPS, UMFPACK, SuperLU and SuperLUDist libraries. After the needed auxiliary packages have been installed, you need to install the library (again from the development branch) from

`https://github.com/sfilippone/mld2p4-2`.

After all these operation have been completed, you can finally install the KINSOL-PSBLAS library from

`https://github.com/Cirdans-Home/kinsol-psblas`.

The installation of this library uses CMake and requires the user to provide the location of the installation directories of PSBLAS, PSBLAS-EXT and MLD2P4-2 libraries.

2 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,

$$A\mathbf{x} = \mathbf{b}, \qquad A \in \mathbb{K}^{n\times n}, \qquad \mathbb{K} = \mathbb{R}, \mathbb{C}, \tag{1}$$

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 $0, \ldots, \mathrm{np}-1$, 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 `basectxt`, 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 $1, \ldots, n$ to a "local" numbering in each process. This means that each process i will own a certain subset $1, \ldots, n_{\mathrm{row}_i}$, each element of which corresponds to a certain element of $1 \ldots n$.

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 $i \in \{1, \ldots, \mathrm{ng}\}$ is assigned to process vg[i]. The vector vg[] must be identical on all calling processes, and its entries have the ranges $(0, \ldots, \mathrm{np}-1)$ or $(1, \ldots, \mathrm{np})$ 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 $(1, \ldots, \mathrm{nl})$ 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

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

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_i_t 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_i_t psb_c_exit(psb_i_t 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_i_t psb_c_abort(psb_i_t ictxt)`

In the next two Sections 3, and Section 4 we describe such encapsulation for dense vector, and sparse matrices. Then in Section 5 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 [4].

3 The NVECTOR_PSBLAS implementation

Library structure

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

! →

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.

3.1 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_OWN_DATA_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)
```

NV_CONTENT_P(v) this macro gives access to the contents of the PSBLAS vector `N_Vector`.

NV_DESCRIPTOR_P(v), NV_OWN_DATA_P(v), NV_PVEC_P(v) these macros give instead individual access to the parts of the content of a PSBLAS parallel `N_Vector`.

NV_ICTXT_P(v) this macro provides the PSBLAS context used by the NVECTOR_PSBLAS vectors.

3.2 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)`

3.2.1 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_i_t m, psb_i_t *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 void N_VSetArrayPointer_PSBLAS(psb_c_dvector *v_data, N_Vector v)

`N_VLinearSum_PSBLAS` : Performs the AXPBY BLAS operation between to NVECTOR_PSBLAS, the result can be both out-of- and in-place.

Prototype void N_VLinearSum_PSBLAS(realtype a, N_Vector x, realtype b, N_Vector y, N_Vector z)

`N_VConst_PSBLAS` : Sets all components of the NVECTOR_PSBLAS to a constant value and assembles it, the user **does not need** to assembly it.

Prototype void N_VConst_PSBLAS(realtype c, N_Vector z)

`N_VProd_PSBLAS` : Performs the entry-wise multiplication of two NVECTOR_PSBLAS, the result can be both out-of- and in-place.

Prototype `void N_VProd_PSBLAS(N_Vector x, N_Vector y, N_Vector z)`

`N_VDiv_PSBLAS` : 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 denumerator. It is up to the user to guarantee that this does not happens.

Prototype `void N_VDiv_PSBLAS(N_Vector x, N_Vector y, N_Vector z)`

`N_VScale_PSBLAS` : Scales an NVECTOR_PSBLAS by a scalar c, the result can be both out-of- and in-place.

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

`N_VAbs_PSBLAS` : Sets the entries of an NVECTOR_PSBLAS to the absolute values of the entries of the input.

Prototype `void N_VAbs_PSBLAS(N_Vector x, N_Vector z)`

`N_VInv_PSBLAS` : 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 `void N_VInv_PSBLAS(N_Vector x, N_Vector z)`

`N_VAddConst_PSBLAS` : Adds a scalar to all components of an NVECTOR_PSBLAS and returns the result in another NVECTOR_PSBLAS object.

Prototype `void N_VAddConst_PSBLAS(N_Vector x, realtype b, N_Vector z)`

`N_VDotProd_PSBLAS` : Compute the dot product of two NVECTOR_PSBLAS objects.

Prototype `realtype N_VDotProd_PSBLAS(N_Vector x, N_Vector y)`

`N_VMaxNorm_PSBLAS` : Compute the max norm of an NVECTOR_PSBLAS object.

Prototype `realtype N_VMaxNorm_PSBLAS(N_Vector x)`

`N_VWrmsNorm_PSBLAS` : Compute the weighted (by the size) 2-norm of an NVECTOR_PSBLAS object.

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

`N_VWrmsNormMask_PSBLAS` : 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 `realtype N_VWrmsNormMask_PSBLAS(N_Vector x,N_Vector w,N_Vector id)`

`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: booleantype 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: booleantype 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)

3.2.2 Fused operation

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

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

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:

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

where c is an array of nv 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 nv other vectors.

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

3.2.3 Vector array operations

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

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

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:

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

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 nv 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 nv 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 nv 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 nv vectors to the corresponding vector in ns 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 ns vector arrays containing nv 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 nvector 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, booleantype tf)
int N_VEnableLinearCombination_PSBLAS(N_Vector v, booleantype
    tf)
int N_VEnableScaleAddMulti_PSBLAS(N_Vector v, booleantype tf)
int N_VEnableDotProdMulti_PSBLAS(N_Vector v, booleantype tf)
int N_VEnableLinearSumVectorArray_PSBLAS(N_Vector v,
    booleantype tf)
int N_VEnableScaleVectorArray_PSBLAS(N_Vector v, booleantype
    tf)
int N_VEnableConstVectorArray_PSBLAS(N_Vector v, booleantype
    tf)
```

```
int N_VEnableWrmsNormVectorArray_PSBLAS(N_Vector v,
    booleantype tf)
int N_VEnableWrmsNormMaskVectorArray_PSBLAS(N_Vector v,
    booleantype tf)
int N_VEnableScaleAddMultiVectorArray_PSBLAS(N_Vector v,
    booleantype tf)
int N_VEnableLinearCombinationVectorArray_PSBLAS(N_Vector v,
    booleantype tf)
```

4 The SUNMATRIX_PSBLAS implementation

Library structure

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

! →

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.

4.1 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 SM_CONTENT_P(A)      ( (SUNMatrixContent_PSBLAS)(A->
    content) )
#define SM_DESCRIPTOR_P(A)   ( SM_CONTENT_P(A)->cdh )
#define SM_PMAT_P(A)         ( SM_CONTENT_P(A)->ah )
#define SM_ICTXT_P(A)        ( SM_CONTENT_P(A)->ictxt )
```

SM_CONTENT_P(A) — this macro gives access to the contents of the PSBLAS matrix `SUNMATRIX`.

SM_DESCRIPTOR_P(A), SM_PMAT_P(A) — these macros give instead individual access to the parts of the content of a PSBLAS parallel `SUNMATRIX`.

SM_CONTENT_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 reinitalization; 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.*

! →

4.2 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 si 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 $i = 1, \ldots, nz$; 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, it 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)`

4.2.1 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 MatrixMarket 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 `void SUNPSBLASMatrix_Print(SUNMatrix A,char *matrixtitle,char *filename )`

`SUNPSBLASMatrix_Rows` : This function returns the number of (global) rows in the PSBLAS SUNMatrix.

Prototype — sunindextype SUNPSBLASMatrix_Rows(SUNMatrix A)

`SUNPSBLASMatrix_Columns` : This function returns the number of (global) columns in the PSBLAS SUNMatrix.

Prototype — sunindextype SUNPSBLASMatrix_Columns(SUNMatrix A)

`SUNPSBLASMatrix_NNZ` : This function returns the (global) number of non-zero entries in the PSBLAS SUNMatrix.

Prototype — sunindextype SUNPSBLASMatrix_NNZ(SUNMatrix A)

`SUNMatGetID_PSBLAS` : Returns the type identifier for the matrix `A`, since this is a custom implementation it returns the value `SUNMATRIX_CUSTOM`.

Prototype — SUNMatrix_ID SUNMatGetID_PSBLAS(SUNMatrix A)

`SUNMatClone_PSBLAS` : Creates a new SUNMatrix on the same descriptor of an existing matrix `A` and sets the `ops` field. It **does not copy** the matrix, but rather allocates storage for the new matrix leaving the new matrix in the BUILD state.

Prototype — SUNMatrix SUNMatClone_PSBLAS(SUNMatrix A)

`SUNMatDestroy_PSBLAS` : 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 — void SUNMatDestroy_PSBLAS(SUNMatrix A)

`SUNMatZero_PSBLAS` : 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 — int SUNMatZero_PSBLAS(SUNMatrix A)

`SUNMatCopy_PSBLAS` : 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 — int SUNMatCopy_PSBLAS(SUNMatrix A, SUNMatrix B)

`SUNMatScaleAdd_PSBLAS` : 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 — int SUNMatScaleAdd_PSBLAS(realtype c, SUNMatrix A, SUNMatrix B)

`SUNMatScaleAddI_PSBLAS` : Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.

Prototype — int SUNMatScaleAddI_PSBLAS(realtype c, SUNMatrix A)

`SUNMatMatvec_PSBLAS` : Performs the matrix-vector product operation, $\mathbf{y} = A\mathbf{x}$. It should only be called with vectors $\mathbf{x}$ and $\mathbf{y}$ 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)

4.2.2 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 a second-order centered finite difference discretization scheme. 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`.

5 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 a version of 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 advanced preconditioners are enable by setting ptype to one of the following string

"AS" : Additive Schwarz (AS) this enables the construction of a one-level domain decomposition preconditioner, based on Additive Schwarz methods for which all the details are described in the ML2DP4 user's and reference guide [2].

"ML" : this enables the construction of a multilevel preconditioner for which details are described in Section 5.3, and are based on the ML2DP4 package [2, 3].

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.

5.1 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_ICTXT_P(S)       ( PSBLAS_CONTENT(S)->ictxt )
#define LS_METHD_P(S)       ( PSBLAS_CONTENT(S)->methd )
#define LS_PTYPE_P(S)       ( PSBLAS_CONTENT(S)->ptype )
```

`PSBLAS_CONTENT(S)` this macro gives access to the contents of the PSBLAS linear solver `SUNLINSOL`.

`LS_PREC_P(S)` this macro gives access to the pointer to the PSBLAS preconditioner object.

`LS_MLPREC_P(S)` this macro gives access to the pointer to the MLD2P4 preconditioner object.

`LS_DESCRIPTOR_P(S)`, `LS_PMAT_P(S)` these macros give instead individual access to the parts of the content of a PSBLAS parallel `SUNMATRIX`.

`LS_ICTXT_P(S)` this macro provides the PSBLAS context used by the SUNMATRIX_PSBLAS sparse matrices and vectors.

`LS_METHD_P(S)` this macro gives access to the string that select the type of Krylov solver to be used as iterative method.

`LS_PTYPE_P(S)` this macro gives access to the string that select the type of preconditioner to be used within the iterative method.

5.2 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. All these calls make sense only after the initialization of the preconditioner by a call to `SUNLinSolInitialize_PSBLAS`; see the next subsection.

`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_i_t 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 5.3, and are fully detailed in [2].

5.2.1 Description of the SUNLINSOL_PSBLAS functions

`SUNLinSol_PSBLAS` : Only the solver options are set at this stage, all the information regarding the communicator, 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[], and the PSBLAS contex on which we are working.

Prototype SUNLinearSolver SUNLinSol_PSBLAS(psb_c_SolverOptions options, char methd [], char ptype[], psb_i_t ictxt)

`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 will be set after that it has been called, i.e., in the case of ptype="ML" preconditioner before 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)

5.3 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., $\Omega = \{1, 2, \ldots, n\}$. 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,

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

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 setup 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 [2] 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 [2] for the full details on how to setup the preconditioner.

5.3.1 An example of linear system solution

We can consider the same context of Example 1, with the matrix construction detail in Section 4.2.2. 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.

6 A complete KINSOL-PSBLAS tutorial

Library structure

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

We consider here a complete example of usage for the KINSOL package with PSBLAS linear algebra and iterative solver routines. Our interest in having a Newton-Krylov solver is mostly related to the task of approximating the solution of nonlinear differential problems. To illustrate the functionalities we have introduced for the KINSOL package we focus on the solution of a model semilinear elliptic partial differential equation in two dimensions, via a finite-difference method.

To arrive at the PDE we wish to solve, we start by considering the problem we are interested in we start by considering the problem of minimizing the energy functional

$$\mathfrak{J}(u) = \int_\Omega \left(\frac{\varepsilon^2}{2} |\nabla u|^2 + g(u) \right), \qquad \Omega = [0,1]^2, \qquad g \in \mathcal{C}^\infty(\mathbb{R}), \tag{3}$$

over a class of admissible *smooth* functions V vanishing on the boundary of the unitary square $\partial\Omega$, i.e., of finding

$$u = \arg\min_{u \in V} \mathfrak{J}(u).$$

Smooth critical points for (3) satisfy the Euler–Lagrange equation

$$\begin{cases} -\varepsilon^2 \nabla^2 u - g'(u) = 0, & \mathbf{x} \in \Omega. \\ u = 0, & \mathbf{x} \in \partial\Omega. \end{cases} \tag{4}$$

To have a closed–form solution to use for comparing the error we consider its forced version

$$\begin{cases} -\varepsilon^2 \nabla^2 u - g'(u) = f, & \mathbf{x} \in \Omega. \\ u = 0, & \mathbf{x} \in \partial\Omega. \end{cases} \tag{5}$$

with

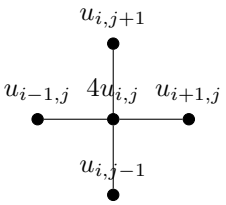
$$\begin{aligned} g' =& u - u^3, \\ f =& e^{-3/\epsilon} \left(4096 \sinh^3\left(\frac{x-1}{2\epsilon}\right) \sinh^3\left(\frac{x}{2\epsilon}\right) \sinh^3\left(\frac{y-1}{2\epsilon}\right) \sinh^3\left(\frac{y}{2\epsilon}\right) \right. \\ & -2e^{2/\epsilon} \left(3\cosh\left(\frac{x-y}{\epsilon}\right) + 3\cosh\left(\frac{x+y-1}{\epsilon}\right) - 2\cosh\left(\frac{1-x}{\epsilon}\right) \right. \\ & \left. \left. -2\cosh\left(\frac{x}{\epsilon}\right) - 2\cosh\left(\frac{1-y}{\epsilon}\right) - 2\cosh\left(\frac{y}{\epsilon}\right) + \cosh\left(\frac{1}{\epsilon}\right) + 1 \right) \right); \end{aligned}$$

so that the function

$$u = \psi(x)\psi/y),$$

is a solution, for

$$\psi(t) = -\exp\left(\frac{t-1}{\epsilon}\right) - \exp\left(-\frac{t}{\epsilon}\right) + \exp\left(-\frac{1}{\epsilon}\right) + 1.$$



Before turning to the implementation, we write down the finite difference formulation of (5). Let us consider the uniform Cartesian grid $\{(x_i = (i-1)h, y_j = (j-1)h)\}_{i,j=0}^{n+1}$ on the unit square for $h = 1/(n+1)$. For the discretization of the Laplace operator in (4) we consider its finite difference approximation on the grid $\{(x_i, y_j)\}$ with a five point stencil, i.e., for $i, j = 2, \ldots, n-1$,

$$\nabla_h^2 u_{i,j} = \frac{4u_{i,j} - u_{i-1,j} - u_{i,j-1} - u_{i+1,j} - u_{i,j+1}}{h^2} + O(h^2).$$

We can then express the the vector algebraic function whose zero approximates the solution to the minimization problem for (3) as

$$F(\mathbf{u}) = \varepsilon^2 A\mathbf{u} - \mathbf{u} + \mathbf{u}^3 - \mathbf{f} = 0. \tag{6}$$

Then, the (generic) Newton iteration for this problem reads as

- Given an initial guess $\mathbf{u}_0 = [1, 1, \ldots, 1]^T$,
- For $k = 1, 2, \ldots$ until convergence (or maximum allotted iterations)
  - Solve for $J_F(\mathbf{u}_k)\mathbf{d}_k = -F(\mathbf{u}_k)$,
  - Update our guess $\mathbf{u}_{k+1} = \mathbf{u}_k + \lambda_k \mathbf{d}_k$.

Where $J_F(\mathbf{u}) = F'(\mathbf{u})$ is the system Jacobian. The "generality" is inherent to a certain number of choices regarding the computation of the step–size λ_k, the update rate for the Jacobian, and so on. This part of the library has not been modified, so we refer back to [1, Chapter 4], and focus instead on the construction of the objects and the routines needed for launching the KINSOL module.

Before, entering into the details regarding the implementation, we compute also a closed form expression of the Jacobian $J_F(\mathbf{c})$ for (6), that reads

$$J_F(\mathbf{u}) = \varepsilon^2 A - I + 3\,\mathrm{diag}(\mathbf{u}^2). \tag{7}$$

As we have discussed in the general framework, the first step is to initialize the PSBLAS environment and to set the distribution of the data. In this case we are going to use also the interoperability between PSBLAS and MPI to use some native MPI functions, thus we recover also the MPI Communicator from the PSBLAS context,

```
psb_i_t        ictxt;      /* PSBLAS Context                */
MPI_Comm       comm;       /* MPI Communicator              */
psb_i_t        np, iam;    /* Number of procs, proc id      */
ictxt = psb_c_init();
psb_c_info(ictxt,&iam,&np);
comm = MPI_Comm_f2c(ictxt);
```

The next step is then deciding the distribution of the data among the processes, this is analogous to what we have seen in the other examples, and can be achieved by

```
psb_c_descriptor *cdh;  /* PSBLAS Descriptor             */
psb_l_t      n,m,nt,nr;
psb_i_t      nlr;

cdh = psb_c_new_descriptor();
psb_c_set_index_base(0);
m = ((psb_l_t) idim )*idim;
n = m;
nt = (m+np-1)/np;
nr = fmax(0,fmin(nt,m-(iam*nt)));
nt = nr;
MPI_Allreduce(MPI_IN_PLACE,&nt,1,MPI_LONG,MPI_SUM,comm);
if(nt != m){
 printf("%d Initialization error %ld %ld %ld\n",iam,nr,nt,m);
 psb_c_barrier(ictxt);
 psb_c_abort(ictxt);
 return(1);
}
if (info=psb_c_cdall_nl(nr, ictxt, cdh)!=0) {
 fprintf(stderr,"From cdall: %d\nBailing out\n",info);
 psb_c_abort(ictxt);
}
nlr = psb_c_cd_get_local_rows(cdh);
```

We are going again for a simple block distribution of the unknowns, in this case another feasible way of performing the data distribution would have been using the MPI_Cart_coords MPI function to achieve a bi-dimensional distribution related to the grid.

The next step is then building the pieces needed to perform the computations of $F(\mathbf{u})$ in (6). In this case the parts that is useful to have computed just one time and to be kept in memory are the matrix $\varepsilon^2 A$ and the forcing term $\mathbf{f}$. While we are at this, we observe also the the Jacobian in (7) shares the same pattern of the A matrix, thus we can also build at the same time the template for it. Moreover, since this is a test problem, we build also the true solution. We use the functions from Section 3, and Section 4 to initialize and allocate the space for the matrix.

```
LAP = NULL;   /* The matrix epsilon^2 A */
LAP = SUNPSBLASMatrix(ictxt, cdh);
J = NULL;     /* The prototype for the Jacobian */
J = SUNPSBLASMatrix(ictxt, cdh);
fvec = NULL; /* The forcing term f */
fvec = N_VNew_PSBLAS(ictxt, cdh);
```

We first compute some useful quantities, namely, the scaling,

```
/* We loop over rows belonging to current process using our
    BLOCK data distribution.                                  */
deltah   = ((psb_d_t) 1.0)/(idim+1);
sqdeltah = deltah*deltah;
deltah2  = ((psb_d_t) 2.0)*deltah;
sizes[0] = idim;
sizes[1] = idim;
```

then we query the communicator to discover what are the global indexes that we own,

```
owned = SUNTRUE;
myidx = (psb_l_t *) malloc( sizeof(psb_l_t)*nlr );
psb_c_cd_get_global_indices(myidx,nlr,owned,cdh);
```

we build an auxiliary matrix consisting of one row at a time; just a small matrix, the procedure might be extended to generate a group of rows per call

```
val = (psb_d_t *) malloc( sizeof(psb_d_t)*20*nb );
irow = (psb_l_t *) malloc( sizeof(psb_l_t)*20*nb );
icol = (psb_l_t *) malloc( sizeof(psb_l_t)*20*nb );
```

Finally we launch the whole assembly procedure as

```
psb_c_barrier(ictxt);
for(int ii = 0; ii < nlr; ii += nb){
  ib = fmin(nb,nlr-ii+1);
  localvecindex = (psb_l_t *) malloc( sizeof(psb_l_t)*ib );
  valj = (psb_d_t *) malloc( sizeof(psb_d_t)*ib );
  icoeff = 0;
  for(int kk = 0; kk < ib; kk++){
    i = ii + kk;
    glob_row = myidx[i]-1;                // Local Matrix
    Pointer
    psb_c_l_idx2ijk(ijk,glob_row,sizes,modes,base);
    x = (ijk[0]+1)*deltah;
    y = (ijk[1]+1)*deltah;

    zt[kk] = f(x,y,epsilon);
    ut[kk] = solution(x,y,epsilon);
    /* Internal point: build discretization
    */
    // term depending on   (x-1,y)
    val[icoeff] = -d(x,y,epsilon)/sqdeltah;
    if (ijk[0] == 0) {
      zt[kk] = g( (psb_d_t) 0, y )*(-val[icoeff]) + zt[kk];
    }else{
      ijktemp[0]    = ijk[0]-1;
      ijktemp[1]    = ijk[1];
      icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,base
    );
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
    }
    // term depending on     (x,y-1)
    val[icoeff] = -d(x,y,epsilon)/sqdeltah;
    if (ijk[1] == 0) {
      zt[kk] = g(x,(psb_d_t) 0)*(-val[icoeff]) + zt[kk];
    }else{
      ijktemp[0]    = ijk[0];
      ijktemp[1]    = ijk[1]-1;
      icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,base
    );
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
    }
    // term depending on     (x,y)
    val[icoeff]  = ( (psb_d_t) 2.0 )*(d(x,y,epsilon) + d(x,y,
    epsilon))/sqdeltah;
```

```
      icol[icoeff] = psb_c_l_ijk2idx(ijk,sizes,modes,base);
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
      //  term depending on     (x,y+1)
      val[icoeff] = -d(x,y,epsilon)/sqdeltah;
      if(ijk[1] == idim - 1){
        zt[kk] = g(x,(psb_d_t) 1.0)*(-val[icoeff]) + zt[kk];
      }else{
        ijktemp[0]   = ijk[0];
        ijktemp[1]   = ijk[1]+1;
        icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,base
      );
        irow[icoeff] = glob_row;
        icoeff       = icoeff + 1;
      }
      // term depending on     (x+1,y)
      val[icoeff] = -d(x,y,epsilon)/sqdeltah;
      if(ijk[0] == idim - 1){
        zt[kk] = g((psb_d_t) 1.0,y)*(-val[icoeff]) + zt[kk];
      }else{
        ijktemp[0]   = ijk[0]+1;
        ijktemp[1]   = ijk[1];
        icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,base
      );
        irow[icoeff] = glob_row;
        icoeff       = icoeff + 1;
      }
    }
    info = SUNMatIns_PSBLAS(icoeff,irow,icol,val,LAP);
    info = SUNMatIns_PSBLAS(icoeff,irow,icol,val,J);
    if(info != 0) printf("%d Error in SUNMatIns! %d\n",iam,info
      );
    icoeff = 0;
    for(int jj=ii; jj < ii+ib; jj++){
      localvecindex[icoeff] = myidx[jj]-1;
      valj[icoeff] = 1.0;
      icoeff++;
    }
    info = psb_c_dgeins(ib,localvecindex,zt,NV_PVEC_P(fvec),
      NV_DESCRIPTOR_P(fvec));
    if(info != 0) printf("%d Error in dgeins! %d\n",iam,info);
    info = psb_c_dgeins(ib,localvecindex,ut,NV_PVEC_P(utrue),
      NV_DESCRIPTOR_P(utrue));
    if(info != 0) printf("%d Error in dgeins! %d\n",iam,info);
    if(info != 0) printf("%d Error in SUNMatIns! %d\n",iam,info
      );
    free(localvecindex);
}
```

Now that the insert phase has been completed we need to assemble both the communicator, the two matrices, and the two vectors.

```
psb_c_cdasb(cdh);
N_VAsb_PSBLAS(utrue);
N_VAsb_PSBLAS(fvec);
SUNMatAsb_PSBLAS(J);
SUNMatAsb_PSBLAS(LAP);
```

Before putting together the KINSOL solver we need

1. the function computing $F(\mathbf{u})$,
2. the function computing $J_F((u))$,
3. an initial guess $\mathbf{u}$.

For this case, as initial guess, we simply select the constant vector with all entries equal to one, namely

```
u = NULL;
```

```
u = N_VNew_PSBLAS(ictxt, cdh);
N_VConst(1.0,u);
```

Then the function for $F(\mathbf{u})$ can be defined as

```
static int funcprpr(N_Vector u, N_Vector fval, void *
    user_data){
struct user_data_for_f *input = user_data;
N_Vector cc;
cc = N_VNew_PSBLAS(NV_ICTXT_P(u),NV_DESCRIPTOR_P(u));
N_VAsb_PSBLAS(cc);
N_VAsb_PSBLAS(fval);
// cc <- u
N_VLinearSum( (psb_d_t) 1.0,u,(psb_d_t) 0.0,cc,cc);
// F(u) = -epsilon^2*A*u - u -f + u^3
SUNMatMatvec_PSBLAS( *input->A, cc, fval );// fval <- Lap cc
N_VLinearSum_PSBLAS( (psb_d_t) -1.0, *input->f, (psb_d_t)
    1.0, fval, fval); // fval = fval - f
N_VLinearSum_PSBLAS( (psb_d_t) -1.0, cc, (psb_d_t) 1.0, fval,
     fval); // fval = fval - cc
N_VProd_PSBLAS(cc,u,cc); // cc^2 = cc.*u
N_VProd_PSBLAS(cc,u,cc); // cc^3 = u.*cc^2
// fval = fval - cc^3
N_VLinearSum_PSBLAS( (psb_d_t) +1.0, cc, (psb_d_t) 1.0, fval,
     fval);
N_VDestroy(cc);
return(0);
}
```

where we are using the auxiliary structure user_data_for_f to store the auxiliary data we need during the assembly, i.e.,

```
struct user_data_for_f {
  SUNMatrix *A;
  N_Vector *f;
  psb_i_t sizes[2];
  psb_d_t sqdeltah;
  psb_d_t epsilon;
};
struct user_data_for_f user_data;/* User data for F,J */
/* We put the precomputed parts in the auxiliary data
    structure, this will be used to make both nonlinear
    function evaluations and Jacobian evaluations */
user_data.A = &LAP;
user_data.f = &fvec;
user_data.sizes[0] = sizes[0];
user_data.sizes[1] = sizes[1];
user_data.sqdeltah = sqdeltah;
user_data.epsilon  = epsilon;
```

In a similar way the routine of the Jacobian is a replica of the procedure we have used to compute A, in which we have changed the term on the diagonal, i.e.,

```
static int jac(N_Vector yvec, N_Vector fvec, SUNMatrix J,
void *user_data, N_Vector tmp1, N_Vector tmp2)
{

  struct user_data_for_f *input = user_data;
  psb_l_t      *myidx,*irow,*icol,glob_row;
  psb_i_t       nlr,ib,icoeff,ijk[2],ijktemp[2],modes=2,base
    =0,i,iam,np,info;
  psb_d_t      *val,x,y, epsilon = input->epsilon, zt[nb];
  psb_d_t      sqdeltah = input->sqdeltah, deltah;
  psb_i_t      *sizes  =  input->sizes;
  psb_i_t      idim = sizes[0];
  bool         owned;
  deltah = sqrt(sqdeltah);
```

```
// Who are we?
psb_c_info(SM_ICTXT_P(J),&iam,&np);
if(iam == 0){
  printf("\tBuilding a new Jacobian\n");
  printf("\tSize of the grid %d x %d\n",sizes[0],sizes[1]);
  printf("\tepsilon = %1.2e deltah = %1.2e\n",epsilon,
  sqdeltah);
}
SUNMatZero(J); // We put to zero the old Jacobian to reuse
  the structure

nlr = psb_c_cd_get_local_rows(SM_DESCRIPTOR_P(J));
owned = SUNTRUE;
myidx = (psb_l_t *) malloc( sizeof(psb_l_t)*nlr );
psb_c_cd_get_global_indices(myidx,nlr,owned,NV_DESCRIPTOR_P
  (yvec));

/* we build an auxiliary matrix consisting of one row at a
  time; just a
small matrix. might be extended to generate a bunch of rows
   per call. */
val = (psb_d_t *) malloc( sizeof(psb_d_t)*20*nb );
irow = (psb_l_t *) malloc( sizeof(psb_l_t)*20*nb );
icol = (psb_l_t *) malloc( sizeof(psb_l_t)*20*nb );

for(int ii = 0; ii < nlr; ii += nb){
  ib = fmin(nb,nlr-ii+1);
  icoeff = 0;
  for(int kk = 0; kk < ib; kk++){
    i = ii + kk;
    glob_row = myidx[i]-1;                // Local Matrix
  Pointer
    psb_c_l_idx2ijk(ijk,glob_row,sizes,modes,base);
    x = (ijk[0]+1)*deltah;
    y = (ijk[1]+1)*deltah;

    zt[kk] = f(x,y,epsilon);
    /* Internal point: build discretization
    */
    // term depending on   (x-1,y)
    val[icoeff] = -d(x,y,epsilon)/sqdeltah;
    if (ijk[0] == 0) {
      zt[kk] = g( (psb_d_t) 0, y )*(-val[icoeff]) + zt[kk];
    }else{
      ijktemp[0]    = ijk[0]-1;
      ijktemp[1]    = ijk[1];
      icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,
  base);
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
    }
    // term depending on     (x,y-1)
    val[icoeff] = -d(x,y,epsilon)/sqdeltah;
    if (ijk[1] == 0) {
      zt[kk] = g(x,(psb_d_t) 0)*(-val[icoeff]) + zt[kk];
    }else{
      ijktemp[0]    = ijk[0];
      ijktemp[1]    = ijk[1]-1;
      icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,
  base);
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
    }
```

Here we have the change with respect to A, and we use the elements of the current solution

```
    // term depending on     (x,y)
    val[icoeff]  = ( (psb_d_t) 2.0 )*(d(x,y,epsilon) + d(x,
  y,epsilon))/sqdeltah
```

```
      -((psb_d_t) 1.0) + ((psb_d_t) 3.0)*pow((
    N_VGetArrayPointer_PSBLAS(yvec))[i],2.0);
      icol[icoeff] = psb_c_l_ijk2idx(ijk,sizes,modes,base);
      irow[icoeff] = glob_row;
      icoeff       = icoeff + 1;
      //  term depending on     (x,y+1)
      val[icoeff] = -d(x,y,epsilon)/sqdeltah;
      if(ijk[1] == idim - 1){
        zt[kk] = g(x,(psb_d_t) 1.0)*(-val[icoeff]) + zt[kk];
      }else{
        ijktemp[0]    = ijk[0];
        ijktemp[1]    = ijk[1]+1;
        icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,
    base);
        irow[icoeff] = glob_row;
        icoeff       = icoeff + 1;
      }
      // term depending on     (x+1,y)
      val[icoeff] = -d(x,y,epsilon)/sqdeltah;
      if(ijk[0] == idim - 1){
        zt[kk] = g((psb_d_t) 1.0,y)*(-val[icoeff]) + zt[kk];
      }else{
        ijktemp[0]    = ijk[0]+1;
        ijktemp[1]    = ijk[1];
        icol[icoeff] = psb_c_l_ijk2idx(ijktemp,sizes,modes,
    base);
        irow[icoeff] = glob_row;
        icoeff       = icoeff + 1;
      }
    }
    info = SUNMatIns_PSBLAS(icoeff,irow,icol,val,J);
    if(info != 0) printf("%d Error in SUNMatIns! %d\n",iam,
    info);
  }

  SUNMatAsb_PSBLAS(J);
  if(iam == 0){
    printf("\tBuilding phase completed\n");
  }

  /* Free the Memory for the Auxiliary Array */
  free(val);
  free(irow);
  free(icol);
  free(myidx);

  return(0);
}
```

Now we have almost all the bits and pieces on the PSBLAS side needed to the define our KINSOL solver, the last thing we need is a vector of constraints than need to be enforced on the Newton iteration, since in this case we have no constraint to enforce this is simply a constant vector of zeros, i.e.,

```
constraints = NULL;
constraints = N_VNew_PSBLAS(ictxt, cdh);
N_VConst(0.0,constraints);
```

and then we define the nonlinear solver as

```
/* Call KINCreate/KINInit to initialize KINSOL:
A pointer to KINSOL problem memory is returned and stored in
    kmem. */
kmem = KINCreate();
info = KINInit(kmem, funcprpr, u);
info = KINSetNumMaxIters(kmem, newtonmaxit);
info = KINSetPrintLevel(kmem, 0);
info = KINSetUserData(kmem, &user_data);
```

```
info = KINSetConstraints(kmem, constraints);
info = KINSetFuncNormTol(kmem, fnormtol);
info = KINSetScaledStepTol(kmem, scsteptol);
/* We no longer need the constraints vector since
    KINSetConstraints creates a private copy for KINSOL to use
    . */
N_VDestroy(constraints);
```

We refer to the guide [1] for the meaning of the different options. Now we need to tell KINSOL how we would like our linear systems with the Jacobian matrix to be solved. We are using a matrix-based approach, so we need to exploit the solver we have introduced in Section 5. Therefore, we build an instance of a PSBLAS solver with a ML preconditioner by doing

```
/* We create now the linear systes solver */
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, methd, ptype, ictxt);
```

Then we set the option for the Multigrid preconditioner as

```
SUNLinSolInitialize_PSBLAS(LS);
info = SUNLinSolSeti_PSBLAS(LS,"SMOOTHER_SWEEPS",2);
info = SUNLinSolSeti_PSBLAS(LS,"SUB_FILLIN",1);
info = SUNLinSolSetc_PSBLAS(LS,"COARSE_SOLVE","BJAC");
info = SUNLinSolSetc_PSBLAS(LS,"COARSE_SUBSOLVE","ILU");
```

We can then the PSBLAS linear solver (together with its preconditioner) to KINSOL by doing

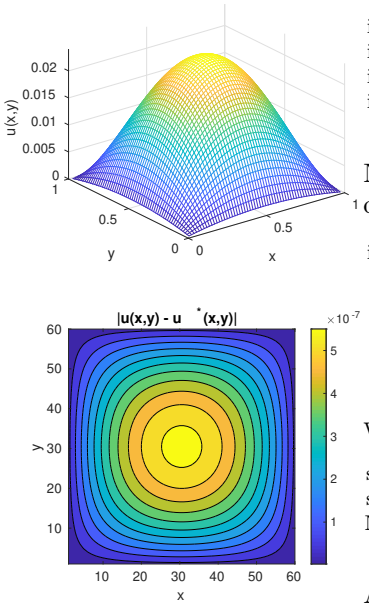
```
info = KINSetLinearSolver(kmem, LS, J);
info = KINSetJacFn(kmem,jac);
info = KINSetEtaForm(kmem,KIN_ETACONSTANT);
info = KINSetEtaConstValue(kmem,options.eps);
```

Now the setup phase is completed, and we have only to call the solver on our problem. This is achieved by doing

```
info = KINSol(kmem,/* KINSol memory block                    */
  u,            /* initial guess on input/solution vector */
  globalstrategy,/* global strategy choice                 */
  sc,     /* scaling vector for the variable u             */
  sc);    /* scaling vector for function values fval       */
```

where the scaling vector is simply

```
sc = NULL;
sc = N_VNew_PSBLAS(ictxt, cdh);
N_VConst(1e-4*sqdeltah,sc);
```



Solution of (6) for a 60×60 grid.

At the end of this call the vector **u** will contain our approximate solution, on the left we have given a plot of the solution of the point-wise difference with the true solution for a 60×60 grid.

The complete code for running the example, and having the possibilities of changing both the parameters of the solvers and of the problems, is contained in `examples/kinsol/psblas`.

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=vl[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*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*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] P. D'Ambra, D. di Serafino, 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.

[3] P. D'Ambra, D. di Serafino, and S. 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.

[4] 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.

[5] Salvatore Filippone and Alfredo Buttari. “Object-oriented techniques for sparse matrix computations in Fortran 2003”. In: *ACM Trans. Math. Software* 38.4 (2012), p. 23.

[6] Salvatore Filippone and Michele Colajanni. “PSBLAS: A library for parallel linear algebra computation on sparse matrices”. In: *ACM Trans. Math. Software* 26.4 (2000), pp. 527–550.