PSFUN

Release 0.1

Fabio Durastante

Oct 20, 2020

CONTENTS:

1 Matrix Functions **3**
1.1 The PSFUN Library 4
1.2 How To Install 4

2 Serial Module **7**
2.1 Module 8

3 Krylov Module **11**
3.1 Stopping Criterion 12
3.2 Module 12

4 Library Usage Examples **15**
4.1 Serial examples 15
4.2 Parallel examples 15

Bibliography **17**

Fortran Module Index **19**

Index **21**

Lo scopo della Matematica è di determinare il valore numerico delle incognite che si presentano nei problemi pratici. Newton, Euler, Lagrange, Cauchy, Gauss, e tutti i grandi matematici sviluppano le loro mirabili teorie fino al calcolo delle cifre decimali necessarie.

—Giuseppe Peano

Non esistono problemi dai quali si può prescindere. Non c'è niente di più penoso di coloro i quali suddividono il pensiero dell'uomo in un pensiero da cui non si può prescindere e in uno da cui si può prescindere. Fra costoro si celano i nostri futuri carnefici.

—Una partita a scacchi con Albert Einstein, Friedrich Dürrenmatt

CHAPTER

1 MATRIX FUNCTIONS

This library is focused on the computation of matrix-function [3] vector products

$$\mathbf{y} = f(A)\mathbf{x}, A \in \mathbb{R}^{n\times n},\ \mathrm{nnz}(A) = O(n),\ f : \mathbb{R} \to \mathbb{R}, \tag{1.1}$$

for large and sparse matrices in a distributed setting. Matrix functions are ubiquitous in models for applied sciences. They are involved in the solution of ordinary, partial, and fractional differential equations, systems of coupled differential equations, hybrid differential-algebraic problems, equilibrium problems, measures of complex networks, and many others.

To perform the computation in (1.1), we consider here two main approaches, the first one makes use of a definition based on the *Cauchy integral* for a matrix function: given a closed contour Γ lying in the region of analyticity of the function $f(x)$ and containing the spectrum of A, $f(A)$ can be defined as

$$f(A) = \frac{1}{2\pi i}\int_{\Gamma} f(z)(zI - A)^{-1}dz. \tag{1.2}$$

By applying a quadrature formula on N points to (1.2), with weights $\{c_j\}_{j=1}^{N}$ and nodes $\{\xi_j\}_{j=1}^{N}$, it is possible to approximate (1.1) as

$$\mathbf{y} = f(A)\mathbf{x} \approx \sum_{j=1}^{N} c_j(A + \xi_j I)^{-1}\mathbf{x},$$

that is then computationally equivalent to the solution of N linear systems with the same right-hand side.

The second approach to problem (1.1) resides instead on the use of projection algorithm. Specifically, we suppose having two k-th dimensional subspaces $\mathcal{V}$ and $\mathcal{W}$ spanned by the column of the matrices $V, W \in \mathbb{R}^{n\times k}$. Then, problem (1.1) can be projected and approximated on the two subspaces by doing

$$\mathbf{y} = f(A)\mathbf{x} \approx W f(V^T A W)V^T\mathbf{x},$$

where now $A_k = V^T AW$ is a small matrix of size $k \times k$, to which we can apply many specific algorithms for the particular choice of $f(x)$, [3], or again a quadrature formula.

1.1 The PSFUN Library

The recent developments on softwares for sparse linear algebra have been made essential for a wide variety of scientific applications. Specifically, they have been dedicated to the construction of of massively parallel sparse solvers for a particular matrix function $f(x) = x^{-1}$, i.e., for the solution of large and sparse linear system. A computational framework that lies at the core of pretty much all multi-physics and multi-scale simulations.

With this library, we try to face the analogous challenge of computing matrix-function vector products for more general functions than the inverse.

The library described here is substantially based on the parallel BLAS feature for sparse matrices made available by the PSBLAS library, and is geared towards the possibility of running on machines with thousands of high-performance cores, and is divided in three main modules,

Serial module: this module implements (or interfaces) the computation of $f(A)$, $f(A)\mathbf{x}$ for matrices of small-size that can be handled in a sequential way,

Krylov module: this module implements distributed Krylov based methods for the reduction of problem (1.1) to the solution of problems of small dimensions,

Quadrature module: this module implements the approach in (1.2) by implementing different quadrature formulas.

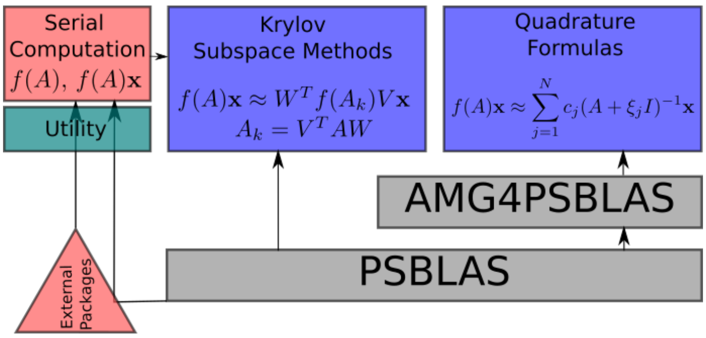

Fig. 1.1: Structure of the PSFUN library.

1.2 How To Install

The first step to install the PSFUN is to obtain and install the PSBLAS library from psctoolkit. All the relevant information can be found there.

The actual version of the library works with the development version of PSBLAS, this can be done obtained via GitHub by doing

```
git clone https://github.com/sfilippone/psblas3.git
cd psblas3
./configure -with-<stuff>=... -prefix=/path/to/psblas
make -j
make install
```

in which the various `-with-<stuff>=...` options can be read from the output of the `./configure -h`, again please refer to the original documentation of PSBLAS for all the relevant information.

Auxiliary packages that can be used to with the library are:

- the package for the computation of φ-functions from [5], that can be obtained from the ACM website.

To build the documentation Sphinx and Sphinx-Fortran (with the relevant dependencies) are needed. Building the documentation is optional, and can be skipped during the configuration phase. In every case a copy of the docs is included with the code.

After having installed all the dependencies, and the auxiliary packages the PSFUN library can be installed via `ccmake` (Version $\geq$ 3.15), by setting the position of PSBLAS, and all the auxiliary packages.

```
git clone https://github.com/Cirdans-Home/psfun.git
mkdir build
cd build
ccmake ../psfun/
make
make install
```

CHAPTER

2 SERIAL MODULE

This module contains the routines needed for the computation of $f(A)x$ for A a matrix of small size. It interfaces external codes and algorithms that usually work with matrix memorized in dense storage. The intended use of the functions contained here is to use them at the lower level of a Krylov subspace method. The library directly contains the EXPOKIT code [10] for the computation of the matrix exponential, together with the scaling and squaring and Taylor algorithms [6][7] by J. Burkardt. For using the φ-functions, the code from [5] is needed. It can be downloaded, compiled and linked to the main library in the install phase.

The module is centered on the `psfun_d_serial` type, this module contains all the options needed to set a specific matrix function to be computed. Not all the options are needed for every type of matrix-function, e.g., the field `integer(psb_ipk_) :: padedegree` is used only if a Padè type algorithm is employed. All the keywords needed to load the implemented functions and algorithic variants are given in Table 2.1.

Table 2.1: Implemented Methods

Function	Variant	Matrix	fname	variant	Source
$f(\alpha A)$	Diagonalization	Symmetric	"USERF"	"SYM"	
$\exp(\alpha A)$	Taylor	General	"EXP"	"TAYLOR"	[6][7]
	Scaling and Squaring	General	"EXP"	"SASQ"	[6][7]
	Generalized Padè	General	"EXP"	"GENPADE"	[10]
	Chebyshev	Hessenberg	"EXP"	"CHBHES"	[10]
	Chebyshev	General	"EXP"	"CHBGEN"	[10]
	Chebyshev	Symmetric	"EXP"	"CHBSYM"	[10]
$\varphi_k(\alpha A)$	Scaling and Squaring	Symmetric	"PHI"	"NONE"	[5]

2.1 Module

Description

This module contains the generic interfaces for the computation of the different matrix functions included in the library. The idea is that this modules computes, in a serial way, $y = f(\alpha A)x$.

Quick access

Types *unknown_type*

Routines *psfun_d_serial_apply_array()*, *psfun_d_serial_apply_sparse()*, *psfun_d_setinteger()*, *psfun_d_setpointer()*, *psfun_d_setreal()*, *psfun_d_setstring()*

Needed modules

- `psb_base_mod`
- `scalesquare`

Types

- `type psfun_d_serial_mod/unknown_type`

  Type fields

  - `% fname` *[character,optional/default='exp']*
  - `% padedegree` *[integer,optional/default=6]*
  - `% phiorder` *[integer,optional/default=1]*
  - `% scaling` *[real,optional/default=1.0_psb_dpk_]*
  - `% variant` *[character,optional/default='expokit']*

Variables

Subroutines and functions

`subroutine psfun_d_serial_mod/psfun_d_setstring(`*fun*, *what*, *val*, *info*`)`

Set function for setting options defined by a string

Parameters

- **fun** :: Function object
- **what** *[character,in]* :: String of option to set
- **val** *[character,in]* :: Value of the string
- **info** *[integer,out]* :: Output flag

Use `psb_base_mod`

`subroutine psfun_d_serial_mod/psfun_d_setreal(`*fun*, *what*, *val*, *info*`)`

Set function for setting options defined by a real

Parameters

- **fun** :: Function object

- **what** *[character,in]* :: String of option to set
- **val** *[real,in]* :: Real Value of the option
- **info** *[integer,out]* :: Output flag

Use `psb_base_mod`

`subroutine psfun_d_serial_mod/psfun_d_setinteger(`*fun*, *what*, *val*, *info*`)`
Set function for setting options defined by an integer

Parameters

- **fun** :: Function object
- **what** *[character,in]* :: String of option to set
- **val** *[integer,in]* :: Integer Value of the option
- **info** *[integer,out]* :: Output flag

Use `psb_base_mod`

`subroutine psfun_d_serial_mod/psfun_d_setpointer(`*fun*, *what*, *val*, *info*`)`
To set the function pointer inside the type

Parameters

- **fun** :: Function object
- **what** *[character,in]* :: String of option to set
- **val** :: Function to set
- **info** *[integer,out]* :: Output flag

Use `psb_base_mod`

`subroutine psfun_d_serial_mod/psfun_d_serial_apply_array(`*fun*, *a*, *y*, *x*, *info*`)`
This is the core of the function apply on a serial matrix to compute $y = f(\alpha * A)x$. It calls on the specific routines implementing the different functions. It is the function to modify if ones want to interface a new function that was not previously available or a new algorithm (variant) for an already existing function.

Parameters

- **fun** :: Function information
- **a** (,) *[real,in]* :: We need to work on a copy of a since the Lapack routine
- **y** (*) *[real,out]* :: Output vector
- **x** (*) *[real,in]* :: Input vector
- **info** *[integer,out]* :: Information on the output

Use `psb_base_mod`, `scalesquare`

`subroutine psfun_d_serial_mod/psfun_d_serial_apply_sparse(`*fun*, *a*, *y*, *x*, *info*`)`
This is the core of the function apply on a serial matrix to compute $y = f(\alpha * A)x$ when A is memorized in a sparse storage. In this case the routine converts it to a dense storage and then calls the array version of itself. That is the one implementing the different functions. It is the function to modify if ones want to interface a new function that was not previously available or a new algorithm (variant) for an already existing function.

Parameters

- **fun** :: Function information
- **a** *[psb_dspmat_type,inout]* :: Matrix
- **y** (*) *[real,out]* :: Output vector
- **x** (*) *[real,in]* :: Input vector

- **info** *[integer,out]* :: Information on the output

Use `psb_base_mod`

CHAPTER

3

KRYLOV MODULE

Let V_k be an orthogonal matrix whose columns $\mathbf{x}_1, \ldots, \mathbf{x}_k$ span an arbitrary Krylov subspace $\mathcal{W}_k(A, \mathbf{x})$ of dimension k. We obtain an approximation of $f(A)\mathbf{x}$ by

$$f(A)\mathbf{x} = V_k f(V_k^T A V_k) V_k^T \mathbf{x}. \tag{3.1}$$

Different methods for the approximation of matrix functions are obtained for different choices of the projection spaces $\mathcal{W}_k(A, \mathbf{x})$.

Given a set of scalars $\{\sigma_1, \ldots, \sigma_{k-1}\} \subset \overline{\mathbb{C}}$ in the the extended complex plane $\overline{\mathbb{C}}$, that are not eigenvalues of A, let

$$q_{k-1}(z) = \prod_{j=1}^{k-1} (\sigma_j - z).$$

The **rational Krylov** subspace of order k associated with A, $\mathbf{x}$ and q_{k-1} is defined by

$$\mathcal{Q}_k(A, \mathbf{x}) = [q_{k-1}(A)]^{-1} \mathcal{K}_k(A, \mathbf{x}),$$

where

$$\mathcal{K}_k(A, \mathbf{x}) = \mathrm{Span}\{\mathbf{x}, A\mathbf{x}, \ldots, A^{k-1}\mathbf{x}\}$$

is the standard polynomial Krylov space.

By defining the matrices

$$C_j = (\mu_j \sigma_j A - I)(\sigma_j I - A)^{-1},$$

where $\{\mu_1, \ldots, \mu_{k-1}\} \subset \overline{\mathbb{C}}$ are such that $\sigma_j \neq \mu_j^{-2}$, it is known that the rational Krylov space can also be written as follows [2]

$$\mathcal{Q}_k(A, \mathbf{x}) = \mathrm{Span}\{\mathbf{x}, C_1\mathbf{x}, \ldots, C_{k-1} \cdots C_2 C_1 \mathbf{x}\}.$$

This general formulation allows to recast most of the classical Krylov methods in terms of a rational Krylov method with a specific choice of σ_j and μ_j. In particular,

- the **polynomial Krylov** method in which $\mathcal{W}_k(A, \mathbf{x}) = \mathcal{K}_k(A, \mathbf{x})$ can be recovered by defining $\mu_j = 1$ and $\sigma_j = \infty$ for each j.

- The **extended Krylov** method [1][4], in which

$$\mathcal{W}_{2k-1}(A,\mathbf{x}) = \operatorname{Span}\{\mathbf{x}, A^{-1}\mathbf{x}, A\mathbf{x}, \ldots, A^{-(k-1)}\mathbf{x}, A^{k-1}\mathbf{x}\},$$

  is obtained by setting

$$(\mu_j, \sigma_j) = \begin{cases} (1,\infty), & \text{for } j \text{ even}, \\ (0,0), & \text{for } j \text{ odd}. \end{cases}$$

- The **shift-and-invert** rational Krylov [8][11], where

$$\mathcal{W}_k(A,\mathbf{x}) = \operatorname{Span}\{\mathbf{x}, (\sigma I - A)^{-1}\mathbf{x}, \ldots, (\sigma I - A)^{-(k-1)}\mathbf{x}\},$$

  is defined by taking $\mu_j = 0$ and $\sigma_j = \sigma$ for each j.

The PSFUN library contains the implementation of several flavour of these methods that can be used for the computation of (3.1), the field in the `psfun_d_krylov` type represent the options neeeded to for setting up and applying the different implemented method for a given matrix function `fun` (represented by an object of type `psfun_d_serial`).

Table 3.1 has the info on the method available.

Table 3.1: Implemented Krylov Methods

Method	Class	Matrix Type	kname	Source
Arnoldi	Polynomial	General	"ARNOLDI"	[9]
Lanczos	Polynomial	Symmetric	"LANCZOS"	[9]

3.1 Stopping Criterion

3.2 Module

Description

The psfun_d_krylov_mod contains the generic call to a Krylov subspace method for the computation of $y = f(A)x$, for A large and sparse.

Quick access

Routines *psfun_d_parallel_apply()*

Needed modules

- `psb_base_mod`
- *psfun_d_serial_mod*: This module contains the generic interfaces for the computation of the different matrix functions included in the library. The idea is that this modules computes, in a serial way, $y = f(\alpha A)x$. . . .

Types

- type psfun_d_krylov_mod/unknown_type

  Type fields

  – % kname *[character,optional/default='arnoldi']*

Variables

Subroutines and functions

`subroutine psfun_d_krylov_mod/psfun_d_setstring(`*meth, what, val, info*`)`

Set function for setting options defined by a string

Parameters

- **meth**
- **what** *[character,in]*
- **val** *[character,in]*
- **info** *[integer,out]*

Use psb_base_mod

`subroutine psfun_d_krylov_mod/psfun_d_parallel_apply(`*meth, fun, a, desc_a, y, x, eps, info*[*, itmax*[*, itrace*[*, istop*[*, iter*[*, err*]]]]]`)`

This is the generic function for applying every implemented Krylov method. The general iteration parameters (like the number of iteration, the stop criterion to be used, and the verbosity of the trace) can be passed directly to this routine. All the constitutive parameters of the actual method, and the information relative to the function are instead contained in the meth and fun objects. The Descriptor object :p psb_desc_type desc_a [in]: Descriptor for the sparse matrix

Parameters

- **meth** :: Krylov method object
- **fun** *[psfun_d_serial,inout]* :: Function object
- **a** *[psb_dspmat_type,in]* :: Distribute sparse matrix
- **y** *[psb_d_vect_type,inout]* :: Output vector
- **x** *[psb_d_vect_type,inout]* :: Input vector
- **eps** *[real,in]* :: Requested tolerance
- **info** *[integer,out]* :: Output flag
- **itmax** *[integer,in,]* :: Maximum number of iteration
- **itrace** *[integer,in,]* :: Trace for logoutput
- **istop** *[integer,in,]* :: Stop criterion
- **iter** *[integer,out,]* :: Number of iteration
- **err** *[real,out,]* :: Last estimate error

Use psb_base_mod, *psfun_d_serial_mod*

CHAPTER

4

LIBRARY USAGE EXAMPLES

4.1 Serial examples

`program serialtest`

Test program for the serial part of the library. This test program loads a matrix from file together with some options to test the serial computation of the matrix functions. Substantially, it test the interfacing with the library doing the serial part.

Use `psb_base_mod`, *psfun_d_serial_mod*, `psb_util_mod` (`mm_mat_read()`, `mm_array_write()`)

4.2 Parallel examples

Polynomial Krylov method examples

`program arnolditest`

Test for the parallel computation of matrix function by means of the psfun_d_arnoldi function. It applies the classical Arnoldi orthogonalization algorithm on a distributed matrix.

Use `psb_base_mod`, *psfun_d_serial_mod*, *psfun_d_krylov_mod*, `psb_util_mod`

BIBLIOGRAPHY

[1] Vladimir Druskin and Leonid Knizhnerman. Extended Krylov subspaces: approximation of the matrix square root and related functions. *SIAM J. Matrix Anal. Appl.*, 19(3):755–771, 1998. URL: https://doi.org/10.1137/S0895479895292400, doi:10.1137/S0895479895292400.

[2] Stefan Güttel. Rational Krylov approximation of matrix functions: numerical methods and optimal pole selection. *GAMM-Mitt.*, 36(1):8–31, 2013. URL: https://doi.org/10.1002/gamm.201310002, doi:10.1002/gamm.201310002.

[3] Nicholas J. Higham. *Functions of matrices*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2008. ISBN 978-0-89871-646-7. Theory and computation. URL: https://doi.org/10.1137/1.9780898717778, doi:10.1137/1.9780898717778.

[4] L. Knizhnerman and V. Simoncini. A new investigation of the extended Krylov subspace method for matrix function evaluations. *Numer. Linear Algebra Appl.*, 17(4):615–638, 2010. URL: https://doi.org/10.1002/nla.652, doi:10.1002/nla.652.

[5] Souji Koikari. Algorithm 894: On a Block Schur–Parlett Algorithm for ϕ-Functions Based on the Sep-Inverse Estimate. *ACM Trans. Math. Softw.*, April 2009. URL: https://doi.org/10.1145/1499096.1499101, doi:10.1145/1499096.1499101.

[6] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Rev.*, 20(4):801–836, 1978. URL: https://doi.org/10.1137/1020098, doi:10.1137/1020098.

[7] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Rev.*, 45(1):3–49, 2003. URL: https://doi.org/10.1137/S00361445024180, doi:10.1137/S00361445024180.

[8] I. Moret and P. Novati. RD-rational approximations of the matrix exponential. *BIT*, 44(3):595–615, 2004. URL: https://doi.org/10.1023/B:BITN.0000046805.27551.3b, doi:10.1023/B:BITN.0000046805.27551.3b.

[9] Y. Saad. Analysis of some Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, 29(1):209–228, 1992. URL: https://doi.org/10.1137/0729014, doi:10.1137/0729014.

[10] Roger B. Sidje. Expokit: A Software Package for Computing Matrix Exponentials. *ACM Trans. Math. Softw.*, 24(1):130–156, March 1998. URL: https://doi.org/10.1145/285861.285868, doi:10.1145/285861.285868.

[11] Jasper van den Eshof and Marlis Hochbruck. Preconditioning Lanczos approximations to the matrix exponential. *SIAM J. Sci. Comput.*, 27(4):1438–1457, 2006. URL: https://doi.org/10.1137/040605461, doi:10.1137/040605461.

FORTRAN MODULE INDEX

p

psfun_d_krylov_mod, 12
psfun_d_serial_mod, 8

INDEX

A

arnolditest (*fortran program*), 15

P

psfun_d_krylov_mod (*module*), 12
psfun_d_parallel_apply() (*fortran subroutine in module psfun_d_krylov_mod*), 13
psfun_d_serial_apply_array() (*fortran subroutine in module psfun_d_serial_mod*), 9
psfun_d_serial_apply_sparse() (*fortran subroutine in module psfun_d_serial_mod*), 9
psfun_d_serial_mod (*module*), 8
psfun_d_setinteger() (*fortran subroutine in module psfun_d_serial_mod*), 9
psfun_d_setpointer() (*fortran subroutine in module psfun_d_serial_mod*), 9
psfun_d_setreal() (*fortran subroutine in module psfun_d_serial_mod*), 8
psfun_d_setstring() (*fortran subroutine in module psfun_d_krylov_mod*), 13
psfun_d_setstring() (*fortran subroutine in module psfun_d_serial_mod*), 8

S

serialtest (*fortran program*), 15

U

unknown_type (*fortran type in module psfun_d_krylov_mod*), 13
unknown_type (*fortran type in module psfun_d_serial_mod*), 8