The Journal of Open Source Software

MatrixFuns.jl: Matrix functions in Julia

Xue Quan [1,2] **and Antoine Levitt** [1]

1 Laboratoire de Mathématiques d'Orsay, Université Paris-Saclay, France **2** School of Mathematical Sciences, Beijing Normal University, China

DOI: 10.21105/joss.08396

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @vlc1
- @shravanngoswamii

Submitted: 05 June 2025
Published: 19 August 2025

Summary

The computation of matrix functions (i.e., $f(A)$ for A a $n \times n$ matrix and $f : \mathbb{C} \to \mathbb{C}$) and their Fréchet derivatives plays a crucial role in many fields of science (Higham, 2008), and in particular in electronic structure calculations within density functional theory and response calculations. For Hermitian A, computing $f(A)$ can be done efficiently and stably by diagonalization. In the non-normal case, however, diagonalization is unstable and alternative schemes have to be used. Even in the Hermitian case, the evaluation of Fréchet derivatives requires (high-order) divided differences, which by Opitz's formula (de Boor, 2005) is equivalent to the exact computation of $f(A)$ for non-normal A.

In this work, we develop `MatrixFuns.jl` a Julia package (Bezanson et al., 2017) to provide the robust computation of matrix functions for arbitrary square matrices and higher-order Fréchet derivatives for Hermitian matrices. This package is tailored towards high accuracy with relatively small matrices and relatively complicated functions f. Our work is based on the Schur-Parlett algorithm (Davies & Higham, 2003; Higham & Al-Mohy, 2010), with the following modifications:

- It supports functions that are discontinuous, or have sharp variations.
- It does not require the computation of arbitrary-order derivatives of f.
- It exploits existing special-purpose methods for computing matrix functions (e.g., for functions involving exponentials or logarithms) when they exist.

Statement of need

`MatrixFuns.jl` aims to provide high-accuracy computations for general matrix functions and arbitrary-order Fréchet derivatives (including divided differences) in Julia. Julia provides some native matrix functions, but the choice is limited to a few functions for which special-purpose algorithms exist (e.g., exponentials, logarithms, matrix powers). There are no dedicated functions in Julia for computing Fréchet derivatives and divided differences; some Julia packages offer tools for their computation (e.g., `ChainRules.jl` (White, 2019), `DFTK.jl` (Herbst et al., 2021)), but are typically limited to first order.

Methods

Matrix functions

The basic principle of the Schur-Parlett algorithm is as follows. First, one performs a Schur decomposition to reduce to the case of an upper triangular matrix. Then, one uses the Parlett recursion, which for a block matrix $A = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$ expresses $B = f(A)$ as $B_{11} = f(A_{11})$, $B_{22} = f(A_{22})$ and B_{12} given by the solution of the Sylvester equation

$A_{11}B_{12} - B_{12}A_{22} = B_{11}A_{12} - A_{12}B_{22}$. In principle, this can be used to compute $f(A)$ by a recursion, but the Sylvester equation becomes ill-conditioned when A_{11} and A_{22} do not have well-separated eigenvalues. In this case, one can use Taylor series, as proposed in Davies & Higham (2003) and Higham & Al-Mohy (2010), but this has the disadvantage of requiring arbitrarily many derivatives of f, which might be impractical in some applications (e.g., when the function is not analytic, or has sharp variations).

Our algorithm attempts to find a partition of the eigenvalues of A (computed using a Schur decomposition) into blocks that are well-separated. The diagonal blocks are then computed using Taylor series, and the Parlett recursion is used to fill out the off-diagonal blocks. The partitioning aims to find small blocks (so that low-order Taylor series can be used) that are well-separated (so that the Parlett recursion is well-conditioned).

To find the partition, we start by partitioning the set of eigenvalues Λ into disjoint clusters Λ_i such that the distance between two such clusters is at least sep, where sep is a user-definable parameter. We then check if the partition is acceptable by estimating the error in all the clusters; if the estimated error is acceptable, we accept the partition; if not, we split the unacceptable clusters further by applying the partitioning algorithm recursively to each unacceptable Λ_i. We estimate the error in a cluster Λ_i of diameter d_i as $\mathrm{err}_i = (\frac{d_i}{\mathrm{scale}})^{\mathrm{max\_deg}+1}$. We accept a cluster if $\mathrm{err}_i < \varepsilon/\mathrm{sep}$. This choice is made to balance the error originating from the Taylor expansion within a cluster err_i with the error incurred by the use of the Parlett recursion ε/sep.

Therefore, our algorithm has the following parameters:

- scale, the characteristic scale of variations of f, set to 1 by default.
- $\mathrm{max\_deg}$, the order of the Taylor series used, which should be set by the user according to the regularity of the function under consideration and the feasibility of computing high-order derivatives (computed automatically using `TaylorSeries.jl` (Benet & Sanders, 2019) and `Arblib.jl` (Dahne, 2025), where the latter is faster in calculating much larger orders and supports some special functions from `SpecialFunctions.jl` (Johnson, 2025)). By default, set to a large value.
- sep, the initial separation distance, set to $0.1 * \mathrm{scale}$ by default following (Davies & Higham, 2003; Higham & Al-Mohy, 2010).
- ε, the target accuracy, set to machine accuracy by default.

In the case where Julia natively supports the computation of $f(A)$ (as determined by trying to compute `f(ones(1,1))` and catching any resulting error), we use them instead of Taylor series to compute diagonal blocks. In the error estimate, we consider $\mathrm{max\_deg} = \infty$, and therefore use a partition with maximal diameter scale. We partition the eigenvalues rather than simply call the native $f(A)$, because f can still have sharp variations, which would cause inaccuracies in $f(A)$. For example:

```
f(x) = I/(I+exp(50*x));

A = [-0.1 10.0 0.0; 0.0 1 5.0; 0.0 0.0 -0.11];

f(A) # native call
3×3 Matrix{Float64}:
 0.993307  -9.03006      -3.33299e7
 0.0        1.92875e-22  -4.48617
 0.0        0.0           0.99593

mat_fun(f, A; scale=1/50) # Schur-Parlett
3×3 Matrix{Float64}:
 0.993307  -9.03006      -28.8619
 0.0        1.92875e-22   -4.48617
 0.0        0.0            0.99593
```

For discontinuous functions, or functions with sharp variations, our algorithm takes as input a color mapping $\mathrm{color} : \mathbb{C} \to \mathbb{Z}, \lambda \mapsto a$, and makes sure that all the eigenvalues inside a cluster have the same color. This ensures that Taylor expansions are not used across the discontinuity boundaries.

Fréchet derivatives

For a Hermitian $A \in \mathbb{C}^{n\times n}$, denote the eigenpairs by $\{(\lambda_i, v_i)\}$. The N-th order Fréchet derivative expresses the variation of $f(A)$ with respect to a set of variations $H_1, \dots, H_N$, and is given by (see the documentation of `MatrixFuns.jl` for details)

$$\mathrm{d}^N f(A) H_1 \cdots H_N = \sum_{i_0,\cdots,i_N=1}^{n} v_{i_0} \left(\sum_{p\in\mathcal{P}_N} (H_{p(1)})_{i_0,i_1} \cdots (H_{p(N)})_{i_{N-1},i_N} \right) f[\lambda_{i_0}, \cdots, \lambda_{i_N}] v_{i_N}^*,$$

where $(H_{p(k)})_{i,j} = v_i^* H_{p(k)} v_j$ and $p \in \mathcal{P}_N$ is an arbitrary permutation of $\{1, \cdots, N\}$. The higher-order divided differences $f[x_0, \dots, x_N]$ defined recursively by

$$f[x_0, \dots, x_N] = \begin{cases} (f[x_0, \dots, x_{N-1}] - f[x_1, \dots, x_N])/(x_0 - x_N), & \text{if } x_0 \neq x_N, \\ \frac{\partial}{\partial z} f[z, x_1, \dots, x_{N-1}]\big|_{z=x_0}, & \text{if } x_0 = x_N. \end{cases}$$

The naive evaluation of this recurrence formula is prone to numerical stabilities. Instead, we compute the divided differences using Opitz's formula

$$f\left(\begin{bmatrix} x_0 & 1 & & \\ & x_1 & \ddots & \\ & & \ddots & 1 \\ & & & x_N \end{bmatrix}\right) = \begin{bmatrix} f[x_0] & f[x_0,x_1] & \cdots & f[x_0,\dots,x_N] \\ & f[x_1] & \ddots & \vdots \\ & & \ddots & f[x_{N-1},x_N] \\ & & & f[x_N] \end{bmatrix}.$$

Therefore, the key point in evaluating the Fréchet derivative reduces to computing matrix functions for upper triangular matrices.

Examples

We first show how to use MatrixFuns.jl to compute the matrix functions, divided differences, and Fréchet derivatives for smooth functions such as exp.

```
using MatrixFuns

A = [-0.1 1.0 0.0; 0.0 -0.05 1.0; 0.0 0.0 0.01];

mat_fun(exp, A) # returns exp(A)
3×3 Matrix{Float64}:
 0.904837  0.92784   0.477323
 0.0       0.951229  0.980346
 0.0       0.0       1.01005

div_diff(exp, -0.1, -0.05, 0.01) # returns exp[-0.1,-0.05,0.01]
0.47732345844677654

H = 0.5 * (A + A'); # generates a Hermitian matrix

hs = map(i -> i * H, [1, 2]);

mat_fun_frechet(exp, H, hs) # returns d^2exp(H)hs[1]hs[2]
3×3 Matrix{Float64}:
```

```
 0.519468  0.347941  0.55445
 0.347941  1.10871   0.46992
 0.55445   0.46992   0.610653
```

In addition to the usual smooth functions, MatrixFuns.jl can also support special functions and discontinuous functions. Here, we use the error function `erf` and the sign function `sign` to show how it can be used to handle functions with different smoothness.

```
using MatrixFuns, SpecialFunctions

A = [-0.1 1.0 0.0; 0.0 -0.05 1.0; 0.0 0.0 0.01];

mat_fun(erf, A) # smooth function
3×3 Matrix{Float64}:
 -0.112463   1.12182   0.0524648
  0.0       -0.056372  1.12759
  0.0        0.0       0.0112834

mat_fun(x -> erf(500x), A; scale=1/500, color=x->x<0 ? 1 : 2) # singular function
3×3 Matrix{Float64}:
 -1.0   0.0  303.03
  0.0  -1.0   33.3333
  0.0   0.0    1.0

mat_fun(sign, A; color=x->Int(sign(x))) # discontinuous function with smooth branches
3×3 Matrix{Float64}:
 -1.0   0.0  303.03
  0.0  -1.0   33.3333
  0.0   0.0    1.0
```

Reference

Benet, L., & Sanders, D. P. (2019). TaylorSeries.jl: Taylor expansions in one and several variables in julia. *Journal of Open Source Software*, *4*(36), 1043. https://doi.org/10.21105/joss.01043

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Dahne, J. (2025). *Arblib.jl*. Zenodo. https://doi.org/10.5281/zenodo.15058432

Davies, P. I., & Higham, N. J. (2003). A Schur-Parlett algorithm for computing matrix functions. *SIAM Journal on Matrix Analysis and Applications*, *25*(2), 464–485. https://doi.org/10.1137/S0895479802410815

de Boor, C. (2005). Divided differences. *Surveys in Approximation Theory*, *1*, 46–69. http://eudml.org/doc/51657

Herbst, M. F., Levitt, A., & Cancès, E. (2021). DFTK: A Julian approach for simulating electrons in solids. *Proc. JuliaCon Conf.*, *3*, 69. https://doi.org/10.21105/jcon.00069

Higham, N. J. (2008). *Functions of Matrices: Theory and Computation*. SIAM. https://doi.org/10.1137/1.9780898717778

Higham, N. J., & Al-Mohy, A. H. (2010). Computing matrix functions. *Acta Numerica*, *19*, 159–208. https://doi.org/10.1017/S0962492910000036

Johnson, S. G. (2025). *SpecialFunctions.jl*. https://github.com/JuliaMath/SpecialFunctions.jl

White, F. (2019). *ChainRules.jl*. Zenodo. https://doi.org/10.5281/zenodo.14926720