Les Cahiers du GERAD ISSN: 0711–2440

`Krylov.jl`: A Julia basket of hand-picked Krylov methods

A. Montoison, D. Orban

G–2022–50

November 2022
Revised: February 2023

`Krylov.jl`: A Julia basket of hand-picked Krylov methods

Alexis Montoison [a]
Dominique Orban [a]

[a] *GERAD and Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Qc), Canada, H3T 2A7*

alexis.montoison@polymtl.ca
dominique.orban@gerad.ca

November 2022
Revised: February 2023
Les Cahiers du GERAD
G–2022–50

Abstract : This paper presents `Krylov.jl`, a Julia package that implements a collection of Krylov processes and methods for solving a variety of linear problems.

Keywords: Julia, numerical linear algebra, Krylov methods, Krylov processes, sparse linear systems, GPU computing

Résumé : Cet article présente `Krylov.jl`, un module Julia qui contient une collection de processus et méthodes de Krylov pour résoudre une variété de problèmes linéaires.

Mots clés : Julia, algèbre linéaire numérique, méthodes de Krylov, processus de Krylov, systèmes linéaires creux, programmation sur GPU

Acknowledgements: Alexis Montoison is supported by an FRQNT grant and an excellence scholarship of the IVADO institute, and Dominique Orban is partially supported by an NSERC Discovery Grant.

1 Summary

Krylov.jl is a Julia (Bezanson et al., 2017) package that implements a collection of Krylov processes and methods for solving a variety of linear problems:

Square systems	Linear least-squares problems	Linear least-norm problems
$Ax = b$	$\min \|b - Ax\|$	$\min \|x\|$ subject to $Ax = b$

Adjoint systems	Saddle-point and Hermitian quasi-definite systems	Generalized saddle-point and non-Hermitian partitioned systems
$Ax = b$ $A^H y = c$	$\begin{bmatrix} M & A \\ A^H & -N \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ c \end{bmatrix}$	$\begin{bmatrix} M & A \\ B & N \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ c \end{bmatrix}$

A^H denotes the conjugate transpose of A. It coincides with A^T, the transpose of A, if A is real. Krylov methods are iterative methods based on (Krylov, 1931) subspaces. They are an alternative to direct methods such as Gaussian elimination or QR decomposition when storage requirements or computational costs become prohibitive, which is often the case for large and sparse linear problems. Contrary to direct methods, which require storing A explicitly, Krylov methods support linear operators to model operator-vector products $u \leftarrow Av$, and in some instances $u \leftarrow A^H w$ because Krylov processes only require those operations to build Krylov subspaces. The same goes with preconditioners, i.e., transformations that modify a linear system into an equivalent form with favorable spectral properties that may yield faster convergence in finite-precision arithmetic. We refer interested readers to (Ipsen and Meyer, 1998) for an introduction to Krylov methods along with (Greenbaum, 1997) and (Saad, 2003) for more details.

2 Statement of need

2.1 Largest collection of Krylov processes and methods

Krylov.jl aims to provide a unified interface for the largest collection of Krylov processes and methods, all programming languages taken together, with six and thirty-three implementations, respectively:

- **Krylov processes**: Arnoldi, Golub-Kahan, Hermitian Lanczos, Montoison-Orban, Non-Hermitian Lanczos, Saunders-Simon-Yip;
- **Krylov methods**: Bicgstab, Bilq, Bilqr, Cg, Cg-lanczos, Cg-lanczos-shift, Cgls, Cgne, Cgs, Cr, Craig, Craigmr, Crls, Crmr, Diom, Dqgmres, Fgmres, Fom, Gmres, Gpmr, Lnlq, Lslq, Lsmr, Lsqr, Minres, Minres-qlp, Qmr, Symmlq, Tricg, Trilqr, Trimr, Usymlq, Usymqr.

MATLAB (MATLAB, 2022) and PETSc (Balay et al., 2022) have eleven and eighteen distinct Krylov methods, respectively. Note that we only consider the number of Krylov methods that generate different iterates without preconditioning. Variants with preconditioning are not counted except it is a flexible one such as Fgmres.

Some processes and methods are not available elsewhere and are the product of our own research. References for each process and method are available in the extensive documentation.

2.2 Support for any floating-point system supported by Julia

Krylov.jl works with real and complex data in any floating-point system supported by Julia, which means that Krylov.jl handles any precision `T` and `Complex{T}` where `T <: AbstractFloat`. Although most personal computers offer IEEE 754 single and double precision computations, new architectures implement native computations in other floating-point systems. In addition, software libraries such as the GNU MPFR, shipped with Julia, let users experiment with computations in variable, extended precision at the software level with the `BigFloat` data type. Working in high precision has obvious benefits in terms of accuracy.

2.3 Support for Nvidia, AMD and Intel GPUs

Krylov methods are well suited for GPU computations because they only require operator-vector products ($u \leftarrow Av$, $u \leftarrow A^H w$) and vector operations ($\|v\|$, $u^H v$, $v \leftarrow \alpha u + \beta v$), which are highly parallelizable. The implementations in Krylov.jl are generic so as to take advantage of the multiple dispatch and broadcast features of Julia. Those allow the implementations to be specialized automatically by the compiler for both CPU and GPU. Thus, Krylov.jl works with GPU backends that build on GPUArrays.jl, including CUDA.jl, AMDGPU.jl and oneAPI.jl, the Julia interfaces to Nvidia, AMD and Intel GPUs.

2.4 Support for linear operators

The input arguments of all Krylov.jl solvers that model A, B, M, N and preconditioners can be any object that represents a linear operator. Krylov methods combined with linear operators allow to reduce computation time and memory requirements considerably by avoiding building and storing matrices. In nonlinear optimization, finding a critical point of a continuous function frequently involves linear systems where A is a Hessian or a Jacobian. Materializing such operators as matrices is expensive in terms of operations and memory consumption and is unreasonable for high-dimensional problems. However, it is often possible to implement efficient Hessian-vector and Jacobian-vector products, for example with the help of automatic differentiation tools.

2.5 In-place methods

All solvers in Krylov.jl have an in-place variant that allows to solve multiple linear systems with the same dimensions, precision and architecture. Optimization methods such as the Newton and Gauss-Newton methods can take advantage of this functionality by allocating workspace for the solve only once. The in-place variants only require a Julia structure that contains all the storage needed by a Krylov method as additional argument. In-place methods limit memory allocations and deallocations, which are particularly expensive on GPUs.

2.6 Performance optimizations and storage requirements

Operator-vector products and vector operations are the most expensive operations in Krylov.jl. We rely on BLAS routines as much as possible to perform those operations. By default, Julia ships with OpenBLAS and provides multithreaded routines. Since Julia 1.6, users can also switch dynamically to other BLAS backends, such as the Intel MKL or BLIS, thanks to the BLAS demuxing library `libblastrampoline`, if an optimized BLAS is available.

A "Storage Requirements" section is available in the documentation to provide the theoretical number of bytes required by each method. Our implementations are storage-optimal in the sense that they are guaranteed to match the theoretical storage amount. The match is verified in the unit tests by way of functions that return the number of bytes allocated by our implementations.

2.7 Examples

Our first example is a simple implementation of the Gauss-Newton method without linesearch for nonlinear least squares. It illustrates several of the facilities of Krylov.jl: solver preallocation and reuse, genericity with respect to data types, and linear operators. Another example based on a simplistic

Newton method without linesearch for convex optimization is also available in the documentation, and illustrates the same concepts in the sections "In-places methods" and "Factorization-free operators".

```
using LinearAlgebra    # Linear algebra library of Julia
using SparseArrays     # Sparse library of Julia
using Krylov           # Krylov methods and processes
using LinearOperators  # Linear operators
using ForwardDiff      # Automatic differentiation
using Quadmath         # Quadruple precision
using MKL              # Intel BLAS

"The Gauss-Newton method for Nonlinear Least Squares"
function gauss_newton(F, JF, x₀::AbstractVector{T}; itmax = 200, tol = √eps(T)) where T
    n = length(x₀)
    x = copy(x₀)
    Fx = F(x)
    m = length(Fx)
    iter = 0
    S = typeof(x)                 # precision and architecture
    solver = LsmrSolver(m, n, S)  # structure that contains the workspace of LSMR
    solved = tired = false
    while !(solved || tired)
        Jx = JF(x)               # Compute J(xₖ)
        lsmr!(solver, Jx, -Fx)   # Minimize ‖J(xₖ)Δx + F(xₖ)‖
        x .+= solver.x           # Update xₖ₊₁ = xₖ + Δx
        Fx_old = Fx              # F(xₖ)
        Fx = F(x)                # F(xₖ₊₁)
        iter += 1
        solved = norm(Fx - Fx_old) / norm(Fx) ≤ tol
        tired = iter ≥ itmax
    end
    return x
end

T = Float128  # IEEE quadruple precision
x₀ = ones(T, 2)
F(x) = [x[1]^4 - 3; exp(x[2]) - 2; log(x[1]) - x[2]^2]         # F(x)
J(y, x, v) = ForwardDiff.derivative!(y, t -> F(x + t * v), 0)  # y ← JF(x)v
Jᵀ(y, x, w) = ForwardDiff.gradient!(y, x -> dot(F(x), w), x)   # y ← JFᵀ(x)w
symmetric = hermitian = false
JF(x) = LinearOperator(T, 3, 2, symmetric, hermitian,
                                           (y, v) -> J(y, x, v),    # non-transpose
                                           (y, w) -> Jᵀ(y, x, w),   # transpose
                                           (y, w) -> Jᵀ(y, x, w))   # conjugate transpose
gauss_newton(F, JF, x₀)
```

Our second example concerns the solution of a complex Hermitian linear system from the SuiteSparse Matrix Collection (Davis and Hu, 2011) with an incomplete Cholesky factorization preconditioner on GPU. The preconditioner P is implemented as an in-place linear operator that performs the forward and backward sweeps with the Cholesky factor to model P^{-1}. Because the system matrix is Hermitian and positive definite, we use the conjugate gradient method. However, other methods for Hermitian systems could be used, including Symmlq, Cr and Minres.

```
using LinearAlgebra                # Linear algebra library of Julia
using SparseArrays                 # Sparse library of Julia
using Krylov                       # Krylov methods and processes
using LinearOperators              # Linear operators
using MatrixMarket                 # Reader of matrices stored in the Matrix Market format
using SuiteSparseMatrixCollection  # Interface to the SuiteSparse Matrix Collection
using CUDA                         # Interface to Nvidia GPUs
using CUDA.CUSPARSE                # Nvidia CUSPARSE library

ssmc = ssmc_db()
```

```
matrices = ssmc_matrices(ssmc, "Bai", "mhd1280b")
paths = fetch_ssmc(matrices, format="MM")
path_A = joinpath(paths[1], "mhd1280b.mtx")
A_cpu = MatrixMarket.mmread(path_A)
m, n = size(A_cpu)
b_cpu = ones(ComplexF64, m)

# Transfer the linear system from the CPU to the GPU
A_gpu = CuSparseMatrixCSR(A_cpu)
b_gpu = CuVector(b_cpu)

# Incomplete Cholesky decomposition LLᴴ ≈ A with zero fill-in
P = ic02(A_gpu, 'O')

# Additional vector required for solving triangular systems
z = similar(CuVector{ComplexF64}, n)

# Solve Py = x
function ldiv_ic0!(P, x, y, z)
  ldiv!(z, LowerTriangular(P), x)   # Forward substitution with L
  ldiv!(y, LowerTriangular(P)', z)  # Backward substitution with Lᴴ
  return y
end

# Linear operator that model the preconditioner P⁻¹
T = ComplexF64
symmetric = false
hermitian = true
P⁻¹ = LinearOperator(T, m, n, symmetric, hermitian, (y, x) -> ldiv_ic0!(P, x, y, z))

# Solve a Hermitian positive definite system with an incomplete Cholesky preconditioner
x, stats = cg(A_gpu, b_gpu, M=P⁻¹)
```

References

S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang. PETSc/TAO users manual. Technical Report ANL-21/39 - Revision 3.18, Argonne National Laboratory, 2022.

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. SIAM Review, 59(1):65–98, 2017.

T. Davis and Y. Hu. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software, 38(1):1–25, 2011.

A. Greenbaum. Iterative methods for solving linear systems. SIAM, 1997.

I. C. Ipsen and C. D. Meyer. The idea behind Krylov methods. The American mathematical monthly, 105(10):889–899, 1998.

A. N. Krylov. On the numerical solution of the equation by which, in technical matters, frequencies of small oscillations of material systems are determined. Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. mat. i estest. nauk, 7(4):491–539, 1931.

MATLAB. version 9.13.0 (R2022b). The MathWorks Inc., Natick, Massachusetts, 2022.

Y. Saad. Iterative methods for sparse linear systems. SIAM, 2003.