libkrylov, a Modular Open-Source Software Library for Extremely Large On-the-Fly Matrix Computations

Dmitrij Rappoport[*,†], Samuel Bekoe[*], Luke Nambi Mohanam[*,‡], Scott Le[*], Naje' George[*], Ziyue Shen[*,§] and Filipp Furche[*,¶]

December 1, 2022

Abstract

We present the design and implementation of *libkrylov*, an open-source library for solving matrix-free eigenvalue, linear, and shifted linear equations using Krylov subspace methods. The primary objectives of libkrylov are flexible API design and modular structure, which enables integration with specialized matrix–vector evaluation "engines". Libkrylov features pluggable preconditioning, orthonormalization, and tunable convergence control. Diagonal (conjugate gradient, CG), Davidson, and Jacobi–Davidson preconditioners are available, along with orthonormal and nonorthonormal (nKs) schemes. All functionality of libkrylov is exposed via Fortran and C application programming interfaces (APIs). We illustrate the performance of libkrylov for eigenvalue calculations arising in time-dependent density functional theory (TDDFT) in the Tamm–Dancoff approximation (TDA) and discuss the convergence behavior as a function of preconditioning and orthonormalization methods.

Keywords: Krylov space methods, iterative algorithms, open-source software, TDHF, TDDFT. ■

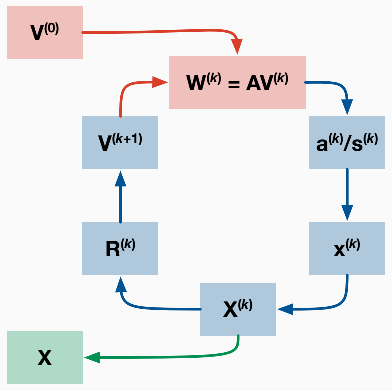
[*]Department of Chemistry, University of California Irvine, Irvine CA, 92697

[†]dmitrij@rappoport.org

[‡]Present address: Department of Electrical and Computer Engineering, Boston University, Boston MA, 02215

[§]Present address: STA Pharmaceutical, San Diego CA, 92121

[¶]filipp.furche@uci.edu



Libkrylov implements iterative Krylov subspace algorithms for eigenvalue, linear, and shifted linear equations within an unified flexible application programming interface (API). The modular structure of libkrylov empowers the users of the library to find the algorithmic features such as preconditioning (conjugate gradient, Davidson, and Jacobi–Davidson) and orthonormalization (orthonormal, nonorthonormal) that best suit their specific use case.

INTRODUCTION

Linear equations and eigenvalue problems with large dimensionality, on the order of $> 10^6$ degrees of freedom, and no special structure or sparsity arise in many scientific and engineering applications, for example, quantum chemical and materials science simulations, partial differential equation solvers, signal reconstruction, and machine learning applications.[1–3] Density functional calculations of light-induced processes in organic and semiconductor nanostructures have pushed the limit of quantum chemical studies to systems with 1000 or more atoms. Molecular and material property calculations in these systems amount to solving eigenvalue problems of dimension one billion or larger.[4,5] In many critical applications, the coefficient matrices are extremely large, dense, and full rank (due to strong coupling/interaction), precluding the use of specialized techniques such as sparse solvers or factorizations.

Matrix-free iterative methods eliminate the bottleneck of explicitly computing and storing extremely large and dense coefficient matrices; instead, products of the coefficient matrix with trial vectors are computed "on the fly". These solvers are based on *Krylov subspace* methods and solve the linear problems such as eigenvalue problems, linear equations, or related problems involving the coefficient matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ or $\in \mathbb{C}^{n\times n}$ on the sequence of Krylov subspaces $\mathcal{K}^{(k)}$ of increasing dimension q_k, $k = 1, 2, \ldots$.[6–12] For many linear problems the iteration converges after K iterations with $q_K \ll n$ and is very economic in its CPU and memory usage even in the case of very large matrices $\mathbf{A}$. The key component of Krylov subspace algorithms is the matrix–vector multiplication "engine", which is capable of efficiently computing $\mathbf{Av}$, given a vector $\mathbf{v}$. These engines are highly application-dependent and often exploit special intermediate representations and/or domain-specific physical properties.[13–22] The importance of Krylov subspace methods for the development of efficient quantum chemical implementations is hard to overstate. Implementations of these methods for molecular property calculations, convergence acceleration of ground-state energy calculations, and other computationally intensive tasks are part of nearly all molecular and solid-state electronic structure codes, many of which were included in a recent software overview.[23] In particular, the Davidson algorithm[24–26] and its variants are behind some of the largest quantum chemical simulations.[27–29] Matrix-free methods are particularly attractive on massively parallel computing architectures, where (re)computing matrix elements is preferable to the storage and communication of the full coefficient matrix $\mathbf{A}$.

While the theory and algorithms of Krylov subspace methods have a wealth of literature[6–12,30,31], the corresponding matrix-free implementations are scattered over many software packages and are often narrowly tailored to the solution of a specific problem. The integration of the optimized matrix–vector multiplication "engines" with the Krylov subspace algorithms and efficient data interchange are crucial to the efficiency of the overall implementation. These requirements rule out general-purpose linear algebra environments such as MATLAB[32] and its open-source clone Octave[33] due to the computational overhead of repeated data transformations. The existing ecosystem of open-source libraries for iterative linear algebra primarily targets problems with an explicit but sparse coefficient matrix $\mathbf{A}$ and does not fit the needs of matrix-free linear algebra problems, for example those arising in molecular property calculations, with regard to their functionality or their data interface. The seminal ARPACK library[34] lacks the implementation of the Davidson algorithm or its variants and has not seen significant development in many years. The popular and efficient BLOPEX library is limited to solving ordinary and generalized eigenvalue problems using the LOBPCG algorithm.[35,36] The SLEPc library is primarily designed for sparse matrix computations but can also accept matrix–vector functions[37,38] instead of explicit matrices. The PETSc library also provides this functionality for solving large partial differential equations.[39]

The many custom implementations of Krylov subspace algorithms with essentially overlapping functionalities resulted in considerable duplication of programming, debugging, and testing effort. Moreover, each implementation is somewhat different in its functionality, configuration options, and numerical thresholds. This results in difficulties in comparing and reproducing results from different codes. Finally, in the absence of well-defined application programming interfaces (APIs), even the existing open-source implementations cannot easily interoperate.

A further complicating factor is that the speed of convergence of iterative methods depends on specific properties of the coefficient matrix, in particular diagonal dominance and the distribution of its eigenvalues. Suitably chosen preconditioning techniques can thus drastically improve convergence of Krylov subspace methods by taking advantage of these properties. In addition, the non-orthonormal Krylov subspace (nKs) approach[5] can exploit the decreasing norms of the vectors added to Krylov subspaces as the iteration progresses to further reduce the computational cost of evaluating matrix–vector products. A related "balancing" approach for the Krylov subspace methods has been proposed.[40] Because the performance characteristics of Krylov subspace methods are structure-dependent, it is desirable to provide the user code with the flexibility to choose the preconditioning and orthonormalization methods best suited for the specific numerical problem.

In this contribution, we describe *libkrylov*[41], an extensible software library for orthogonal and non-orthogonal matrix-free Krylov subspace methods, which features flexible APIs and enables pluggable preconditioning, orthonormalization, and tunable convergence control. The key design objective of libkrylov is to balance the integration of the optimized matrix–vector multiplication and the Krylov subspace algorithms with the simplicity and flexibility of use. Libkrylov is designed as a framework, which inverts the control flow compared to that of typical libraries. While library functions are designed to be called from a driver procedure, the libkrylov solver function is responsible for executing the control flow, calling the user-provided matrix–vector product function in each iteration via a fixed API. This approach ensures the necessary flexibility of the libkrylov components without compromising the efficiency of the implementation, while the user code is only responsible for implementing a matrix–vector multiplication function and can focus on domain-specific optimizations. The libkrylov library is implemented in portable Fortran 2003/C99 and aims to be an off-the-shelf library component for large-scale scientific and engineering applications. To this end, libkrylov is designed with maximal modularity, structured Fortran and C interfaces, and integration with platform-optimized BLAS[42–44] and LAPACK[45] linear algebra primitives in mind. Libkrylov is distributed under the open source 3-clause BSD license.

The version 1.1.0 of libkrylov treats symmetric eigenvalue problems, linear equations, and shifted linear equations. Diagonal (conjugate-gradient, CG) preconditioning, Davidson preconditioner[24–26], and Jacobi–Davidson preconditioning due to Sleijpen and van der Vorst[46,47] can be selected for convergence acceleration. Both the most commonly used orthonormal Krylov subspace algorithm and the computationally efficient nKs approach of Furche and co-workers[5] are available. Moreover, libkrylov supports blocked algorithms for simultaneous iteration of multiple equations with a shared coefficient matrix for linear problems with block-diagonal structure. *A posteriori* error bounds and dynamic restart capability cover the most common application scenarios.

This paper describes the design strategy of libkrylov and outlines its approach to solving a wide range of matrix-free linear problems via a common function-based interface. The structure of the paper is as follows. We first introduce the notation and give a brief overview of Krylov subspace algorithms implemented in libkrylov. After that, we describe the design and implementation of libkrylov. Finally, using an illustrative set of linear problems related to molecular property calculations, we review the suite of preconditioning and orthonormalization techniques available in libkrylov and give examples of their convergence.

KRYLOV SUBSPACE METHODS

Background and Notation

In this section we establish our notation for Krylov subspace methods for eigenvalue problems, linear equations, and shifted linear equations for symmetric coefficient matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$ or Hermitian matrices in $\mathbb{C}^{n\times n}$. Since many applications seek $p \geq 1$ simultaneous solutions involving the same coefficient matrix $\mathbf{A}$, for example, the p lowest eigenvalues, right-hand sides (RHS) and/or shifts, we present the Krylov subspace algorithms in their blocked form and use matrix notation throughout. Table 1 gives an overview of the linear problems treated here. $\mathbf{X}$ denotes the $n \times p$ matrix having solution vectors as columns. For eigenvalue problems, $\mathbf{X}$ is the matrix of eigenvectors. The corresponding eigenvalues are represented by the diagonal matrix $\mathbf{\Omega} = \mathrm{diag}(\Omega_1, \ldots, \Omega_p)$. The RHS of the (shifted) linear equations are combined into the $n \times p$ matrix $\mathbf{P}$. Shifted linear equations additionally contain a matrix of diagonal shifts $\boldsymbol{\omega} = \mathrm{diag}(\omega_1, \ldots, \omega_p)$. These equations can be considered as the diagonal variant of the Sylvester equations.[48]

In Krylov subspace methods, the k-th iterates ($k \geq 1$) are written as vectors in the

Krylov subspace $\mathcal{K}^{(k)}$ of dimension q_k,

$$\mathbf{X}^{(k)} = \mathbf{V}^{(k)}\mathbf{x}^{(k)}, \tag{1}$$

where $\mathbf{V}^{(k)}$ is the matrix of basis vectors of $\mathcal{K}^{(k)}$ and $\mathbf{x}^{(k)}$ are the expansion coefficients. The residual (error) vectors of the k-th iterates are collected in the matrix $\mathbf{R}^{(k)}$, see Table 1 for definitions. The k-th iterates are chosen according to the Ritz–Galerkin condition, that is, the residual vectors are required to be orthogonal to the Krylov subspace $\mathcal{K}^{(k)}$,

$$\mathbf{V}^{(k)\dagger}\mathbf{R}^{(k)} = \mathbf{0}. \tag{2}$$

Note that superscript † indicates the matrix transpose for real matrices and the Hermitian conjugate in the complex case. In this case, $\mathbf{X}^{(k)}$ are the Ritz vectors, and the corresponding expansion coefficients $\mathbf{x}^{(k)}$ follow from solving the projected linear problem on $\mathcal{K}^{(k)}$, as shown in Table 1. The nKs method[5] uses a nonorthonormal subspace basis, in which case the Gram (overlap) matrix of the subspace basis is given by

$$\mathbf{s}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{V}^{(k)} \tag{3}$$

The Rayleigh matrix

$$\mathbf{a}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{A}\mathbf{V}^{(k)} \tag{4}$$

is the projection of the matrix $\mathbf{A}$ onto the Krylov subspace. In eigenvalue problems, the diagonal matrix $\mathbf{\Omega}^{(k)}$ contains the k-th eigenvalue iterates. The projection of the RHS matrix $\mathbf{P}$ of (shifted) linear equations is denoted as

$$\mathbf{p}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{P} \tag{5}$$

The columns of the residual matrix $\mathbf{R}^{(k)}$ are used to construct the basis of the subsequent Krylov subspace $\mathcal{K}^{(k+1)}$ of dimension $q_{k+1} = q_k + p$. The basic Krylov subspace iteration in libkrylov (no preconditioner) adds the residuals to the Krylov subspace basis, similar to Arnoldi iteration.[49] However, most applications apply a suitably chosen invertible $n \times n$ matrix preconditioning matrix $\mathbf{K}^{(k)}$ to the residuals,

$$\tilde{\mathbf{R}}^{(k)} = \left(\mathbf{K}^{(k)}\right)^{-1}\mathbf{R}^{(k)}, \tag{6}$$

prior to expanding the Krylov subspace. Preconditioning techniques are crucial to the performance of Krylov subspace methods for linear equations.[6,11,50–53] In eigenvalue problems, Davidson[24,26] and Jacobi–Davidson (JD) preconditioning[46,47] are widely used convergence acceleration techniques.

Krylov Subspace Algorithm

Given the $q_1 \times n$ matrix of starting vectors $\mathbf{V}^{(1)}$ and matrix–vector multiplication function $f(\mathbf{V}) = \mathbf{AV}$, the Krylov subspace algorithm takes the steps outlined in Fig. 1. Note that matrix–vector products and other quantities involving the columns $\mathbf{V}^{(k)}$ can be reused from previous iterations so that $f(\mathbf{V})$ is only evaluated for new vectors. The step denoted by (*) is only needed for nonorthonormal subspace bases, see below. In the orthonormal case, the modified Gram–Schmidt (MGS) method[48] is applied in the step (‡). The projection (†) is performed for (shifted) linear equations only.

Preconditioning

The convergence of the basic Krylov subspace iteration is often unsatisfactory for practical calculations. The choice of the preconditioning matrix $\mathbf{K}$ can significantly affect the speed of convergence.[6–8,10,11,30,31] The diagonal approximation $\mathbf{D} = \mathrm{diag}(A_{11}, \ldots, A_{nn})$ to the coefficient matrix,

$$\mathbf{K}_{\mathrm{CG}}^{(k)} = \mathbf{D}\,, \tag{7}$$

is often used in the preconditioned CG algorithm.[6,7,11] For eigenvalue problems, the Davidson algorithm corresponds to choosing the preconditioner as

$$\mathbf{K}_{\mathrm{D}}^{(k)} = \mathbf{D} - \boldsymbol{\Omega}^{(k)}\,, \tag{8}$$

where $\boldsymbol{\Omega}^{(k)}$ are the k-th eigenvalue iterates.[24,26] For shifted linear equations, the equivalent of the Davidson preconditioner includes the diagonal shifts $\boldsymbol{\omega}$ instead of eigenvalues.

The JD algorithm ensures by projection that the preconditioned residuals are orthogonal to the solution,[46,47]

$$\mathbf{K}_{\mathrm{JD}}^{(k)} = (\mathbf{1} - \Pi^{(k)})\mathbf{K}_{\mathrm{D}}^{(k)}(\mathbf{1} - \Pi^{(k)})\,, \tag{9}$$

where $\Pi^{(k)}$ is the projector onto the set of k-th iterates. By imposing orthogonality constraints, the preconditioned residuals are computed as

$$\tilde{\mathbf{R}}^{(k)} = \left(\mathbf{K}_\mathrm{D}^{(k)}\right)^{-1}\mathbf{R}^{(k)} - \varepsilon^{(k)}\left(\mathbf{K}_\mathrm{D}^{(k)}\right)^{-1}\mathbf{X}^{(k)} \quad \text{with} \quad \varepsilon^{(k)} = \frac{\mathbf{X}^{(k)\dagger}\left(\mathbf{K}_\mathrm{D}^{(k)}\right)^{-1}\mathbf{R}^{(k)}}{\mathbf{X}^{(k)\dagger}\left(\mathbf{K}_\mathrm{D}^{(k)}\right)^{-1}\mathbf{X}^{(k)}}. \tag{10}$$

In the case of $p > 1$, two variants of the JD preconditioner can be formulated. Each residual vector can be preconditioned by projecting out only the corresponding solution (JD variant 1) or all solutions (JD variant 2).

Nonorthonormal Subspace Bases

The nKs method takes advantage of the fact that the residual norms $\mathbf{R}^{(k)}$ usually decrease as the approximate solutions $\mathbf{X}^{(k)}$ converge to the true solution $\mathbf{X}$. If the function $f(\mathbf{V}) = \mathbf{AV}$ can be made to execute more efficiently for small $\|\mathbf{V}\|$, for example, by prescreening the matrix elements of $\mathbf{A}$, then the cost of each iteration may be reduced by up to 80%[5]. In order to preserve the decreasing residual norms, the orthonormalization step (‡) is omitted in the nKs method, and instead the projected equations are solved in their generalized form with a Gram matrix $\mathbf{s}^{(k)} \neq \mathbf{1}$ (Table 1).

The projected equations are solved in two steps. First, the equation is scaled by the diagonal $\mathbf{d}$ of $\mathbf{s}^{(k)}$,

$$\mathbf{d} = \mathrm{diag}(s_{11}^{(k)}, \ldots, s_{q_k q_k}^{(k)}) \tag{11}$$

to reduce the condition number of $\mathbf{s}$. Using the Cholesky decomposition of the scaled Gram matrix

$$\mathbf{LL}^\dagger = \mathbf{d}^{-1/2}\mathbf{s}^{(k)}\mathbf{d}^{-1/2} \tag{12}$$

the following substitutions can be made

$$\begin{aligned}\tilde{\mathbf{a}}^{(k)} &= \mathbf{L}^{-1}\mathbf{d}^{-1/2}\mathbf{a}^{(k)}\mathbf{d}^{-1/2}(\mathbf{L}^{-1})^\dagger,\\ \tilde{\mathbf{x}}^{(k)} &= \mathbf{L}^\dagger\mathbf{d}^{1/2}\mathbf{x}^{(k)},\\ \tilde{\mathbf{p}}^{(k)} &= \mathbf{L}^{-1}\mathbf{d}^{-1/2}\mathbf{p}^{(k)}.\end{aligned} \tag{13}$$

The resulting linear problem can be then solved using standard methods. For more details, see Ref.[5].

If $p > 1$ equations are simultaneously iterated, the (preconditioned) residuals corresponding to different solutions may become linearly dependent. To improve the condition of the Gram matrix while maintaining the advantage of decreasing residual norms of the nKs approach, we test a variant of the nKs approach. In this *semiorthonormal* variant, instead of the full orthonormalization (‡) in Fig. 1, we limit ourselves to enforcing orthogonality between the columns of $\tilde{\mathbf{R}}^{(k)}$ using a singular-value decomposition (SVD),

$$\begin{aligned} \mathbf{u}^{(k)\dagger}\boldsymbol{\sigma}^{(k)}\mathbf{v}^{(k)} &= \left(\mathbf{K}^{(k)}\right)^{-1}\mathbf{R}^{(k)}\,, \\ \tilde{\mathbf{R}}^{(k)} &= \boldsymbol{\sigma}^{(k)}\mathbf{v}^{(k)}\,. \end{aligned} \tag{14}$$

For $p = 1$, the semiorthonormal variant is identical to the nKs method.[5]

Error Bounds and Convergence of the Algorithm

The norm of the residual matrix $\mathbf{R}^{(k)}$ provides a strict upper bound for the error of the approximate eigenvalues in the k-th iteration[48],

$$|\Omega_i - \Omega_i^{(k)}| \leq \sqrt{2}\|\mathbf{R}^{(k)}\|_2 \quad \text{for } 1 \leq i \leq q_k\,, \tag{15}$$

where the 2-norm of the residual matrix is used. Because

$$\|\mathbf{R}^{(k)}\|_2 \leq \|\mathbf{R}^{(k)}\|_F \leq p\, r_{\max}^{(k)} \quad \text{with} \quad r_{\max}^{(k)} = \max\nolimits_{1\leq i\leq p}|\mathbf{R}_i^{(k)}|\,, \tag{16}$$

the maximum residual norm yields a convenient convergence criterion, $r_{\max}^{(k)} \leq \tau$, where τ is the convergence threshold.

An alternative convergence measure for Krylov subspace algorithms monitors the change in the corresponding Lagrangian functional F. The latter is a quadratic form whose stationary points are the solutions of the linear problem.[5] For eigenvalue problems, the Lagrangian takes the form

$$F[\mathbf{X}, \boldsymbol{\Omega}] = \left\langle \mathbf{X}^\dagger\mathbf{A}\mathbf{X} - \boldsymbol{\Omega}\left(\mathbf{X}^\dagger\mathbf{X} - \mathbf{1}\right)\right\rangle\,, \tag{17}$$

where $\langle\cdot\rangle$ denotes the trace operation. The minimum of $F[\mathbf{X}, \boldsymbol{\Omega}]$ over all $n \times p$ matrices is given by the matrix containing the lowest p eigenvectors as columns. The Lagrangian functional for (shifted) linear equations is given by

$$F[\mathbf{X}] = \left\langle \mathbf{X}^\dagger\mathbf{A}\mathbf{X} - \boldsymbol{\omega}\left(\mathbf{X}^\dagger\mathbf{X} - \mathbf{1}\right) - \mathbf{X}^\dagger\mathbf{P} - \mathbf{P}^\dagger\mathbf{X}\right\rangle\,, \tag{18}$$

in which linear equation is obtained by settings the shifts to zero, $\boldsymbol{\omega} = \mathbf{0}$. If all shifts are smaller than lowest eigenvalue of $\mathbf{A}$, the quadratic form is convex, and the stationary point is a minimum.

In the case of a restart after k iterations, the Krylov subspace basis is replaced by the set of the p solution iterates $\mathbf{X}^{(k)}$. In (shifted) linear problems, the columns of $\mathbf{X}^{(k)}$ must be first orthonormalized before proceeding.

LIBKRYLOV DESIGN AND IMPLEMENTATION

The principal objective of libkrylov is serving as a flexible computational framework, which supports a wide variety of matrix-free linear problems including eigenvalue problems, linear equations, and shifted linear equations via a simple, uniform API. The user should be empowered to evaluate different preconditioning and orthonormalization techniques and experiment with hybrid approaches as demanded by the application domain. Composability is thus a central consideration. Since scientific software development is split between Fortran and C/C++ ecosystems, libkrylov aims to be interoperable with both language families and to provide for simple cross-platform build and installation procedures.

To make good on these promises, libkrylov uses a layered architecture, in which an object-oriented core is wrapped in a function-based interface layer, which allows calling from both Fortran and C/C++ (Fig. 2). The library is implemented in portable Fortran 2003 and takes advantage of the runtime polymorphism capabilities afforded by this standard. However, in the design of libkrylov, composition is preferred over inheritance as it induces looser coupling between components and allows for better composability. The simplified class diagram of libkrylov is shown in Fig. 3. The central component of libkrylov is the `space_t` abstract class, which represents a general Krylov subspace and provides an interface to the steps of the iterative algorithm of Fig. 1. The implementations of the Krylov subspace algorithm for the real symmetric and complex Hermitian cases largely follow analogous paths. It would thus be desirable to use generic programming techniques to abstract over the underlying arithmetic. However, due to the lack of support for generic programming in Fortran 2003, one has to emulate generic classes by polymorphic types such as `real_space_t` and `complex_space_t` inheriting from `space_t`. Such *ad hoc* polymorphism might seem an unappealing alternative because it requires a significant amount of code duplication. But polymorphic classes also offer several advantages as they make extensions to other cases easy, for example for structured problems. For simplicity, we focus on the function of the `real_space_t` class and its components in the following. However, the details apply analogously to the complex case. The generalization of the libkrylov code to different floating-point precisions (single, double, and potentially quadruple) is implemented using code preprocessing, which is not part of the Fortran 2003 standard but is nevertheless almost universally supported by Fortran compilers.

The internal structure of the `real_space_t` class is designed to maximize the separation of concerns between the equation type, orthonormalization, preconditioning, and convergence control. Therefore, the `real_space_t` class contains polymorphic components for the equation type, orthonormalizer, and preconditioner, see Fig. 3. The current implementation supports eigenvalue problems, linear equations, and shifted linear equations. The orthonormal Krylov subspace method, the nKs method, and the semiorthonormal variant are implemented via a common orthonormalizer interface. The preconditioning choices include the null preconditioner (no preconditioning, analogous to Arnoldi iteration), diagonal preconditioning (CG preconditioner), Davidson and Jacobi–Davidson (JD) preconditioning techniques.

This design of the `real_space_t` class allows the user to freely mix and match the components of the Krylov subspace algorithm when constructing a new Krylov subspace object. Flexible convergence control and transparent error handling enable experimentation to determine the best setup depending on the properties of the specific problem. The convergence control object tracks the norm of the residual matrix, the condition number of the overlap matrix (for a nonorthonormal basis), and the Lagrangian of the linear problem over the course of the iterative process. Different combinations of convergence and restart criteria can thus be specified by configuration, while sensible default settings are provided by the library.

The requirement of interoperability with both Fortran and C/C++ code bases significantly constrains the structure of the libkrylov interface. A simple function-based API was chosen for user interaction from both Fortran and C/C++. All `space_t` objects are created as global module variables inside the libkrylov library and addressed using numerical indices. Data exchange between user code and library code is done incrementally with separate function calls, for example, for starting vectors, RHS, or diagonal shifts. This approach has the advantage that new equation types can be added without breaking backwards compatibility. Examples of libkrylov calls from Fortran and C are shown in Figs. 4 and 5.

The user interaction with libkrylov consists of only a few function calls. At minimum, the library must be initialized by the `(c)krylov_initialize` function, the problem type and the relevant dimensions must be specified in the `(c)krylov_add_space` function call, and the initial subspace basis must be passed to the `(c)krylov_set_real_space_vectors` function. The type of arithmetic (real or complex), equation type, and matrix structure are specified by character constants, in analogy to BLAS. In our example, `'r'` indicates real arithmetic, `'e'` means eigenvalue problem, and `'s'` stands for a symmetric coefficient matrix. The bulk of the computation takes place within the `(c)krylov_solve_real_equation` call. The matrix–vector product function `multiply` is passed as the first argument. For the sake of simplicity, the evaluation of matrix–vector products is simulated in Figs. 4 and 5 by multiplication with an explicit coefficient matrix. In real-life applications, the corresponding matrix–vector product function is provided by the user and may be arbitrarily complex as long as it adheres to the interface. The convergence or any error conditions of the iterative algorithm are communicated to the user by the return value of the `(c)krylov_solve_real_equation` function call, with 0 indicating success. In that case, the solution vectors may be retrieved by the `(c)krylov_get_real_space_solutions`. The convergence may be verified by checking the residual norm of the last iteration (`(c)krylov_get_space_last_residual_norm`). The converged eigenvalues may be obtained by additional function calls, which are omitted for brevity. After the interaction with libkrylov is completed, the allocated memory is freed by the (`(c)krylov_finalize`) function call. The complete libkrylov API documentation is provided with the libkrylov code.[41]

The current implementation keeps all program data in memory, which allows for the fastest data access. However, this strategy becomes unfeasible for very large problems or in memory-constrained environments. For these cases, a flexible data storage backend is planned for future releases, which enables both in-memory and out-of-core algorithms with adaptive blocking.

For ease of installation, distribution, and testing, libkrylov uses the CMake cross-platform build system[54]. The library is released under the open-source 3-clause BSD license.

NUMERICAL TESTS

The computational cost and the speed of convergence of Krylov subspace algorithms may strongly depend on the numerical properties of the coefficient matrix $\mathbf{A}$ and the details of the iterative procedure, in particular, on the choice of the preconditioning and orthonormalization methods. The evaluation of matrix–vector products in each iteration is the most computationally expensive step, scaling as $\mathcal{O}(n^2)$ for general dense matrices, as long as the number of iterations $K \ll n$. A large literature is devoted to specialized matrix–vector multiplication "engines", which can achieve significant computational cost reductions per iteration by implementing domain-specific intermediate representations, prescreening of matrix elements of $\mathcal{A}$, and other structure-dependent techniques.[5,13–22,40] In this section, we give selected examples of usage of libkrylov in calculations of molecular property calculations by time-dependent Hartree–Fock (TDHF)[55,56] and time-dependent density functional theory (TDDFT) methods.[57–61] For illustration purposes, we limit ourselves to calculations of size $n = 10^3..10^4$, in which the properties of the coefficient matrix $\mathbf{A}$ can be computed in advance and correlated with the performance characteristics of Krylov subspace algorithms. We focus on the convergence behavior of the iterative algorithm as a function of the preconditioning and orthonormalization techniques. We consider the structural characteristics contributing to the efficiency of the Davidson preconditioner and related methods. The efficiency of iterative algorithms for solving the linear and eigenvalue equations of TDHF and TDDFT is compared in Ref.[62].

Within the Tamm–Dancoff approximation (TDA)[63–65] to TDDFT and the configuration interaction singles (CIS) method,[66] which may be considered the equivalent approximation to TDHF, electronic excitation energies and intensities are obtained from the solutions of a real symmetric eigenvalue equation,

$$\mathbf{A}\mathbf{X}_i = \Omega_i \mathbf{X}_i \,, \tag{19}$$

where the coefficient matrix $\mathbf{A}$ is positive definite. In most cases, only the $i = 1, ..., p$ lowest eigenvalues are of interest. Static polarizabilities in TDHF and TDDFT are computed by solving the real symmetric linear equation

$$(\mathbf{A}+\mathbf{B})\,(\mathbf{X}+\mathbf{Y})_\alpha = (\mathbf{P}+\mathbf{Q})_\alpha \,, \tag{20}$$

for the Cartesian components $\alpha = x, y, z$. The RHS vectors $(\mathbf{P}+\mathbf{Q})_\alpha$ contains matrix elements of the dipole moment operator.

Table 2 summarizes the structural characteristics of the coefficient matrices and the speed of convergence of eigenvalue problems for selected medium-size inorganic and organic molecules with TDDFT using the PBE exchange–correlation functional[67] and the resolution-of-the-identity (RI-J) approximation for the Coulomb interaction.[17,68] The basis sets used in the calculations were of split-valence quality (def2-SVP) and triple-zeta valence quality (def2-TZVP),[69] including diffuse augmentation (def2-SVPD).[70] $p = 1, 2$, and 10 lowest eigenvalues were determined simultaneously. The residual norm convergence criterion was $\tau = 10^{-7}$. The coefficient matrices and RHS vectors were generated by a modified version of the TURBOMOLE[71] escf program.

For each coefficient matrix $\mathbf{A}$, selected numerical characteristics relevant to the convergence and numerical stability of Krylov subspace algorithms are shown. Specifically, Table 2 includes the inverse 2-norm condition number κ_2^{-1},

$$\kappa_2 = \|\mathbf{A}\|_2 \|\mathbf{A}^{-1}\|_2 \,, \tag{21}$$

which serves as a general measure of numerical stability in operations involving $\mathbf{A}$. The ratio ρ_p of the highest requested eigenvalue and the next-higher eigenvalue controls the convergence of iterative algorithms for the p lowest eigenvalues in the absence of preconditioning,[9,72]

$$\rho_p = \Omega_{p+1}/\Omega_p \,. \tag{22}$$

To illustrate the performance of preconditioners based on the diagonal approximation to $\mathbf{A}$, Table 2 shows the average eigenvalue shift of the p lowest eigenvalues $\overline{\delta\Omega}_p$ of $\mathbf{A}$ (in a.u.), which characterizes the the closeness of the spectra of $\mathbf{A}$ and its diagonal approximation $\mathbf{D}$,

$$\overline{\delta\Omega}_p = \frac{1}{p}\sum_{i=1}^{p} \min_{j=1}^{n} |\Omega_i - A_{jj}| \,. \tag{23}$$

The average of the p lowest eigenvalues $\overline{\Omega}_p$ (in a.u.) is included in for comparison. Lastly, Table 2 lists iteration counts K of the orthonormal Krylov subspace algorithm until convergence (within maximum residual norm threshold $\tau = 10^{-7}$) using $q_0 = p$ starting vectors. Results are given for the CG preconditioner, Davidson (D) preconditioner, and Jacobi–Davidson preconditioner, variants 1 (JD1) and 2 (JD2). Without preconditioning, the Krylov subspace algorithm does not convergence within 50 iterations and is thus of little use for the eigenvalue problems considered here.

As Table 2 shows, the coefficient matrices are well-conditioned with $\kappa_2^{-1} = 10^{-4}..10^{-2}$, even with relatively flexible and diffuse basis sets. The spectral radius of $\mathbf{A}$ increases slowly with basis set size, as excitations at the high end of the spectrum are progressively added due to the greater flexibility of the basis set. At the same time, the low end of the spectrum is relatively dense, as shown by the eigenvalue ratios ρ_p close to 1. In some cases, for example, S_8 ($p = 10$) and C_{20} ($p = 2$), the p-th lowest eigenvalue is part of a generate set due to point group symmetry, which results in the eigenvalue ratio ρ_p being exactly 1. The small eigenvalue gap explains the slow convergence in the absence of preconditioning. However, the preconditioning techniques based on the matrix diagonal lead to fast convergence in all cases. Notably, the Davidson preconditioner and the JD1 and JD2 variants of the Jacobi–Davidson preconditioner show nearly identical iteration counts and are superior to the CG preconditioning. Fig. 6 illustrates the convergence of the maximum residual norms in trans-thioindigo for $p = 1, 2$, and 10 lowest excitations. As expected from the small eigenvalue gap, the unpreconditioned Krylov subspace algorithm, which is equivalent to Arnoldi iteration, converges very slowly, managing to only reduce the maximum residual norm by only about one order of magnitude after 50 iterations. Diagonal (CG) preconditioning achieves convergence in 34 iterations for the lowest eigenvalue, 27 iterations for the $p = 2$ lowest eigenvalues, and 23 iterations for $p = 10$ lowest eigenvalues. With the Davidson and Jacobi–Davidson preconditioners, the algorithm converges in 23 iterations for the lowest eigenvalue, 19 iterations for the $p = 2$ lowest eigenvalues, and 14 iterations for the

$p = 10$ lowest eigenvalue. Alternatively, the convergence of the Krylov subspace algorithm can be assessed using the corresponding Lagrangian functional F, as shown in Fig. 7. Once a sufficiently large subspace is constructed, the Lagrangian converges very rapidly towards its stationary value F_0.

Fig. 8 compares the convergence of the orthonormal Krylov subspace algorithm with the nKs and the semiorthonormal methods for the lowest excitations in trans-thioindigo. As discussed in detail in Ref. 5, the nonorthonormal algorithms do not degrade the speed of convergence but are helpful in reducing the computational effort per iteration, if the matrix–vector product function can exploit the reduction in vector norms during the iterative process. In our tests, the semiorthonormal variant has not shown an improvement over the nKs method.

The dependence of the speed of convergence on the dimension of the starting subspace basis is illustrated in Fig. 9 for calculations in trans-thioindigo using the Davidson preconditioner. Larger starting subspace produce faster convergence due to greater overlap with the desired eigenvectors. With smaller starting subspaces, monotonic convergence is often achieved only after one or more "jumps", during which the maximum residual norm is increasing. However, the final subspace dimension q_K at convergence is not strongly dependent on q_0: In the example of Fig. 9, the lowest eigenvalue is converged for $q_K = 23, 24, 21, 25$, and 31 given starting subspace dimensions of $q_0 = 1, 2, 4, 8, 16$, respectively. Similarly, in the case of $p = 2$ lowest eigenvalues, the final subspace dimension is $q_K = 35, 32, 37, 39$ for $q_0 = 2, 4, 8, 16$, respectively. For $p = 10$ lowest eigenvalues, the final subspace dimension is $q_K = 107, 104, 109$ for $q_0 = 10, 12$, and 16, respectively. The optimal choice of the starting subspace dimension depends on the specifics of the matrix–vector product computation. If the computational effort for simultaneously computing q matrix–vector products is proportional to q, one should seek to minimize the final subspace dimension q_K. In that case, $q_0 = p + 2..4$ appears to give the best results for the systems considered in this work. If the computational effort is approximately independent of q, as is the case in integral-direct TDDFT and TDHF implementations,[58,60,61] one prefers to minimize the number of iterations K and should choose a somewhat larger starting subspace. In this work, the choice of $q_0 = p + 6..8$ yields the smallest number of iterations. However, we should stress that the choice of the initial subspace may be strongly system-dependent, and a preliminary study is advisable for picking a suitable initial dimension. Testing different choices for the initial dimension can also help detect potential "missing" eigenvectors in the presence of symmetries.

The good performance of the Davidson method and related algorithm is frequently linked to the diagonal dominance of the coefficient matrix $\mathbf{A}$.[26,46] The matrix $\mathbf{A}$ is called diagonally dominant if the following condition is satisfied:[48,73]

$$|A_{ii}| \geq r'_i \quad \text{for all } i = 1..n, \quad \text{where} \quad r'_i = \sum_{j \neq i} |A_{ij}| \,. \tag{24}$$

The Gershgorin circle theorem relates the eigenvalues of $\mathbf{A}$ to the magnitude of the off-diagonal elements,

$$|\Omega_i - A_{jj}| \leq r'_j \quad \text{for some} \quad j = 1..n \,, \tag{25}$$

that is each eigenvalue of $\mathbf{A}$ lies within a circle in the complex plane with the center at A_{jj} and the radius r'_j for some j. Crucially, the radius of the Gershgorin circle is determined by the off-diagonal elements. The eigenvalues of diagonally dominant matrices must thus be close to some diagonal elements. The converse statement, does not need to hold, however, as we will see shortly.

If we apply the definition of diagonal dominance to the smallest diagonal element of the matrix $\mathbf{A}$ for trans-thioindigo, we observe that this matrix is far from being diagonally dominant in the sense of Eq. (24). For example, the smallest diagonal value of the matrix $\mathbf{A}$, is approximately 0.086 a.u., while the corresponding Gershgorin circle sweeps the sizeable interval $[-0.880, 1.052]$, indicating large off-diagonal elements by absolute value. While the Gershgorin disks contain the eigenvalues of $\mathbf{A}$, the corresponding bounds are too loose to be useful. Similar observations can be made for the other coefficient matrices considered here. However, Table 2 shows that despite not being diagonally dominant, the coefficient matrices nevertheless show only relatively small deviations $\overline{\delta\Omega}_p$ between the eigenvalues of the coefficient matrix $\mathbf{A}$ and its diagonal values at the low end of the spectrum. With the exception of C_{20}, the eigenvalue shift $\overline{\delta\Omega}_p$ is 20% or less of the average eigenvalue $\overline{\Omega}_p$. The small eigenvalue shift explains the superior performance of the Davidson method since the preconditioning matrix $\mathbf{K}_{\mathrm{D}}^{(k)}$ is near-singular and enhances the relevant eigenvector in a way similar to inverse iteration.[48]

The performance of Krylov subspace algorithms depends significantly on the properties of the coefficient matrix $\mathbf{A}$. While the numerical tests discussed here are illustrative of the problems encountered in molecular property calculations, in other cases the above conclusions need not hold. In this case, the user may well want to benchmark the preconditioners on their specific problem. The modular structure of libkrylov makes this benchmarking straightforward. We note that in our numerical tests, the Jacobi–Davidson method does not offer an improvement over the Davidson preconditioner. A similar observation has been previously reported for nearly diagonal matrices.[74] However, the Jacobi–Davidson preconditioner can give better performance for matrices that are not nearly diagonal, especially if a more accurate approximation to $\mathbf{A}$ than its diagonal $\mathbf{D}$ is chosen as the preconditioning matrix. For these matrices, the comparison with the LOBPCG method[35,36] is also instructive.

CONCLUSIONS

Libkrylov strives to unify both battle-tested and more recent algorithmic developments in on-the-fly matrix computations within an extensible framework that enables simple integration with user codes and easy experimentation. The scientific software developer does not need to reinvent the wheel and can focus on the domain-specific aspects of implementing an efficient matrix–vector product function. The parameters of the Krylov subspace algorithm, such as orthonormalization and preconditioning methods, can be then optimized for the specific application. As an open-source library, the continuing development of libkrylov is driven by the requirements of its users. Due to its modular structure and a flexible API, implementations of Krylov subspace algorithms for different equation types, intermediate representations, and preconditioners can be incorporated in an efficient and backwards-compatible manner. More algorithms and implementations from the rich landscape of Krylov subspace methods[6–11,30,31] may be contributed to libkrylov in the future.

Several directions for further developments are currently envisioned. Structured problems of the symplectic type[5] arise in the computation of response properties and excitation energies with TDHF[55,56] and TDDFT methods.[57–61] The extension to linear problems involving real unsymmetric and complex non-Hermitian coefficient matrices is necessary for implementations of coupled cluster (CC) response theory.[75,76] An unsymmetric variant of the Davidson preconditioner[77] can be used in these cases. Finally, additional equation types, for example, Sylvester equation and least-squares problems allow for expansions into other problem domains.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (NSF) under Grant OAC-1835909. S. B. was supported by a fellowship from The Molecular Sciences Software Institute (MolSSI) under NSF grant CHE-2136142. The authors thank Dr. Jonathan Moussa of MolSSI for helpful discussions. Additional Supporting Information may be found in the online version of this article.

References

1. F. Alexander, A. Almgren, J. Bell, A. Bhattacharjee, J. Chen, P. Colella, D. Daniel, J. DeSlippe, L. Diachin, E. Draeger, et al., Philos. Trans. R. Soc., A **378**, 20190056 (2020).

2. T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M. S. Shephard, A. Abdelfattah, et al., Int. J. High Perform. Comput. Appl. **35**, 527 (2021).

3. J. Dongarra, L. Grigori, and N. J. Higham, Philos. Trans. R. Soc., A **378**, 20190066 (2020).

4. S. Kilina, D. Kilin, and S. Tretiak, Chem. Rev. **115**, 5929 (2015).

5. F. Furche, B. T. Krull, B. D. Nguyen, and J. Kwon, J. Chem. Phys. **144**, 174105 (2016).

6. Y. Saad, *Iterative Methods for Sparse Linear Systems* (SIAM, Philadelphia, 2003), 2nd ed., ISBN 9780-89871-53-4-7.

7. H. A. van der Vorst, *Iterative Krylov Methods for Large Linear Systems* (Cambridge University Press, Cambridge, 2003), ISBN 978-0-52118-370-3.

8. D. S. Watkins, *The Matrix Eigenvalue Problem. GR and Krylov Subspace Methods* (SIAM, Philadelphia, 2007), ISBN 978-0-89871-641-2.

9. Y. Saad, *Numerical Methods for Large Eigenvalue Problems* (SIAM, Philadelphia, 2011), ISBN 9781-611-9707-2-2.

10. J. Liesen and Z. Strakoš, *Krylov Subspace Methods* (Oxford University Press, Oxford, 2012).

11. M. A. Olshanskii and E. E. Tyrtyshnikov, *Iterative Methods for Linear Systems* (SIAM, Philadelphia, 2014).

12. C. Huang, W. Liu, Y. Xiao, and M. R. Hoffmann, J. Comput. Chem. **38**, 2481 (2017), ISSN 1096-987X.

13. B. Roos, Chem. Phys. Lett. **15**, 153 (1972).

14. J. Almlöf, K. Faegri, and K. Korsell, J. Comput. Chem. **3**, 385 (1982).

15. M. Häser and R. Ahlrichs, J. Comput. Chem. **10**, 104 (1989).

16. H. Weiss, R. Ahlrichs, and M. Häser, J. Chem. Phys. **99**, 1262 (1993).

17. R. Bauernschmitt, M. Häser, O. Treutler, and R. Ahlrichs, Chem. Phys. Lett. **264**, 573 (1997).

18. D. Rappoport and F. Furche, J. Chem. Phys. **122**, 064105 (2005).

19. F. Neese and G. Olbrich, Chem. Phys. Lett. **362**, 170 (2002).

20. T. Petrenko, S. Kossmann, and F. Neese, J. Chem. Phys. **134**, 054116 (2011).

21. C. Ko, D. K. Malick, D. A. Braden, R. A. Friesner, and T. J. Martínez, J. Chem. Phys. **128**, 104103 (2008).

22. T. M. Maier, H. Bahmann, and M. Kaupp, J. Chem. Theory Comput. **11**, 4226 (2015).

23. C. D. Sherrill, D. E. Manolopoulos, T. J. Martínez, and A. Michaelides, J. Chem. Phys. **153**, 070401 (2020).

24. E. R. Davidson, J. Comput. Phys. **17**, 87 (1975).

25. B. Liu, in *Numerical Algorithms in Chemistry: Algebraic Methods*, edited by C. Moler and I. Shavitt (Lawrence Berkeley Laboratory, Berkeley, 1978), pp. 49–53.

26. M. Crouzeix, B. Philippe, and M. Sadkane, SIAM J. Sci. Comput. **15**, 62 (1994).

27. S. Tretiak, C. M. Isborn, A. M. N. Niklasson, and M. Challacombe, J. Chem. Phys. **130**, 054111 (2009).

28. D. Zuev, E. Vecharynski, C. Yang, N. Orms, and A. I. Krylov, J. Comput. Chem. **36**, 273 (2015).

29. K. D. Vogiatzis, D. Ma, J. Olsen, L. Gagliardi, and W. A. de Jong, J. Chem. Phys. **147**, 184111 (2017).

30. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, Philadelphia, 1994).

31. Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems. A Practical Guide* (SIAM, Philadelphia, 2000).

32. *MATLAB version 9.12.0.1884302 (R2022a)*, The Mathworks, Inc., Natick, Massachusetts (2022).

33. J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, *GNU Octave version 7.3.0 manual: a high-level interactive language for numerical computations* (2022), URL `https://octave.org/doc/v7.3.0/`.

34. R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide* (SIAM, Philadelphia, 1998).

35. A. V. Knyazev, SIAM J. Sci. Comput. **23**, 517 (2001).

36. A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov, SIAM J. Sci. Comput. **29**, 2224 (2007).

37. V. Hernandez, J. E. Roman, and V. Vidal, ACM Trans. Math. Softw. **31**, 351 (2005).

38. E. Romero and J. E. Roman, ACM Trans. Math. Softw. **40**, 13 (2014).

39. E. Bueler, *PETSc for Partial Differential Equations: Numerical Solutions in C and Python* (SIAM, 2020), ISBN 978-161-197-6-30-4.

40. R. M. Parrish, E. G. Hohenstein, and T. J. Martínez, J. Chem. Theory Comput. **12**, 3003 (2016).

41. F. Furche and co workers, *libkrylov, a modular open-source software library for extremely large eigenvalue and linear problems*, `https://gitlab.com/libkrylov/libkrylov-stable` (2022).

42. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, ACM Trans. Math. Softw. **5**, 308 (1979).

43. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Trans. Math. Softw. **14**, 1 (1988).

44. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, ACM Trans. Math. Softw. **16**, 1 (1990).

45. E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., *LAPACK Users' Guide* (SIAM, Philadelphia, 1999).

46. G. L. G. Sleijpen and H. A. van der Vorst, SIAM Rev. **42**, 267 (2000).

47. M. E. Hochstenbach and Y. Notay, GAMM-Mitteilungen **29**, 368 (2006).

48. G. H. Golub and C. F. Van Loan, *Matrix Computations* (Johns Hopkins University Press, Baltimore, 2013), 4th ed.

49. W. E. Arnoldi, Quart. Appl. Math. **9**, 17 (1951).

50. O. Axelsson, BIT Numer. Math. **25**, 165 (1985).

51. A. M. Bruaset, *A Survey of Preconditioned Iterative Methods* (CRC Press, Boca Raton, 1995).

52. M. Benzi, J. Comput. Phys. **182**, 418 (2002).

53. A. J. Wathen, Acta Numer. **24**, 329 (2015).

54. Kitware, Inc., *CMake*, https://cmake.org (2021).

55. A. McLachlan and M. Ball, Rev. Mod. Phys. **36**, 844 (1964).

56. D. J. Thouless, *The Quantum Mechanics of Many-Body Systems* (Academic Press, New York, 1972), 2nd ed.

57. M. E. Casida, in *Recent Advances in Density Functional Methods*, edited by D. E. Chong (World Scientific, Singapore, 1995), vol. 1 of *Recent Advances in Computational Chemistry*, chap. 5, pp. 155–192.

58. R. Bauernschmitt and R. Ahlrichs, Chem. Phys. Lett. **256**, 454 (1996).

59. F. Furche, J. Chem. Phys. **114**, 5982 (2001).

60. R. E. Stratmann, G. E. Scuseria, and M. J. Frisch, J. Chem. Phys. **109**, 8218 (1998).

61. F. Furche and D. Rappoport, in *Computational Photochemistry*, edited by M. Olivucci (Elsevier, Amsterdam, 2005), chap. III, pp. 93–128.

62. J. Kauczor, P. Jørgensen, and P. Norman, J. Chem. Theory Comput. **7**, 1610 (2011).

63. I. Tamm, in *Selected Papers*, edited by B. M. Bolotovskii, V. Y. Frenkel, and R. Peierls (Springer, 1991), chap. N.4, pp. 157–174, 1st ed., ISBN 9783-64274-62-8-4.

64. S. M. Dancoff, Phys. Rev. **78**, 382 (1950).

65. S. Hirata and M. Head-Gordon, Chem. Phys. Lett. **314**, 291 (1999).

66. J. B. Foresman, M. Head-Gordon, J. A. Pople, and M. J. Frisch, J. Phys. Chem. **96**, 135 (1992).

67. J. P. Perdew, K. Burke, and M. Ernzerhof, Phys. Rev. Lett. **77**, 3865 (1996).

68. K. Eichkorn, O. Treutler, H. Öhm, M. Häser, and R. Ahlrichs, Chem. Phys. Lett. **242**, 652 (1995).

69. F. Weigend and R. Ahlrichs, Phys. Chem. Chem. Phys. **7**, 3297 (2005).

70. D. Rappoport and F. Furche, J. Chem. Phys. **133**, 134105 (2010).

71. S. G. Balasubramani, G. P. Chen, S. Coriani, M. Diedenhofen, M. S. Frank, Y. J. Franzke, F. Furche, R. Grotjahn, M. E. Harding, C. Hättig, et al., J. Chem. Phys. **152**, 184107 (2020).

72. B. N. Parlett, *The Symmetric Eigenvalue Problem* (SIAM, Philadelphia, 1998), ISBN 0-89871-402-8.

73. R. A. Horn and C. R. Johnson, *Matrix Analysis* (Cambridge University Press, Cambridge, 2012), 2nd ed., ISBN 978-05215-4-823-6.

74. Y. Notay, SIAM J. Matrix Anal. Appl. **26**, 522 (2004), ISSN 0895-4798.

75. H. Sekino and R. J. Bartlett, Int. J. Quantum Chem. **26**, 255 (1984), ISSN 0020-7608.

76. H. Koch and P. Jørgensen, J. Chem. Phys. **93**, 3333 (1990).

77. K. Hirao and H. Nakatsuji, J. Comput. Phys. **45**, 246 (1982), ISSN 0021-9991.

```
Choose $q_1 \geq p$ starting vectors $\mathbf{V}^{(1)}$
for $k = 1, 2, \ldots$
    $\mathbf{W}^{(k)} = f(\mathbf{V}^{(k)}) = \mathbf{A}\mathbf{V}^{(k)}$
    $\mathbf{s}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{V}^{(k)}$ (*)
    $\mathbf{a}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{W}^{(k)}$
    $\mathbf{p}^{(k)} = \mathbf{V}^{(k)\dagger}\mathbf{P}$ (†)
    Compute $\mathbf{x}^{(k)}$ from projected equation (see Table 1)
    $\mathbf{X}^{(k)} = \mathbf{V}^{(k)}\mathbf{x}^{(k)}$
    Compute residuals $\mathbf{R}^{(k)}$ (see Table 1)
    if $\max_{1 \leq i \leq p}|\mathbf{R}_i^{(k)}| \leq \tau$ then quit
    $\tilde{\mathbf{R}}^{(k)} = \mathbf{K}^{(k)-1}\mathbf{R}^{(k)}$
    Orthogonalize $\tilde{\mathbf{R}}^{(k)}$ against $\mathbf{V}^{(k)}$ (‡)
    $\mathbf{V}^{(k+1)} = [\mathbf{V}^{(k)} \;\; \tilde{\mathbf{R}}^{(k)}]$
end
```

Figure (1) Schematic Krylov subspace iteration algorithm including preconditioning and orthonormalization. $f(\mathbf{V}^{(k)})$ is the matrix–vector product evaluation by the user-supplied function. The step denoted by (*) is only used in nonorthonormal Krylov subspace algorithm. The orthonormal case includes the vector orthonormalization step (‡). The projection (†) is performed for (shifted) linear equations only. τ is the convergence threshold.

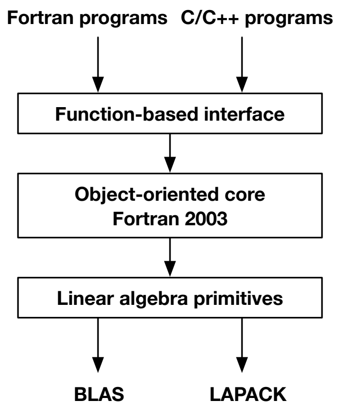


Figure (2) Libkrylov architecture. Invocations are indicated by arrows, composition by inclusion.

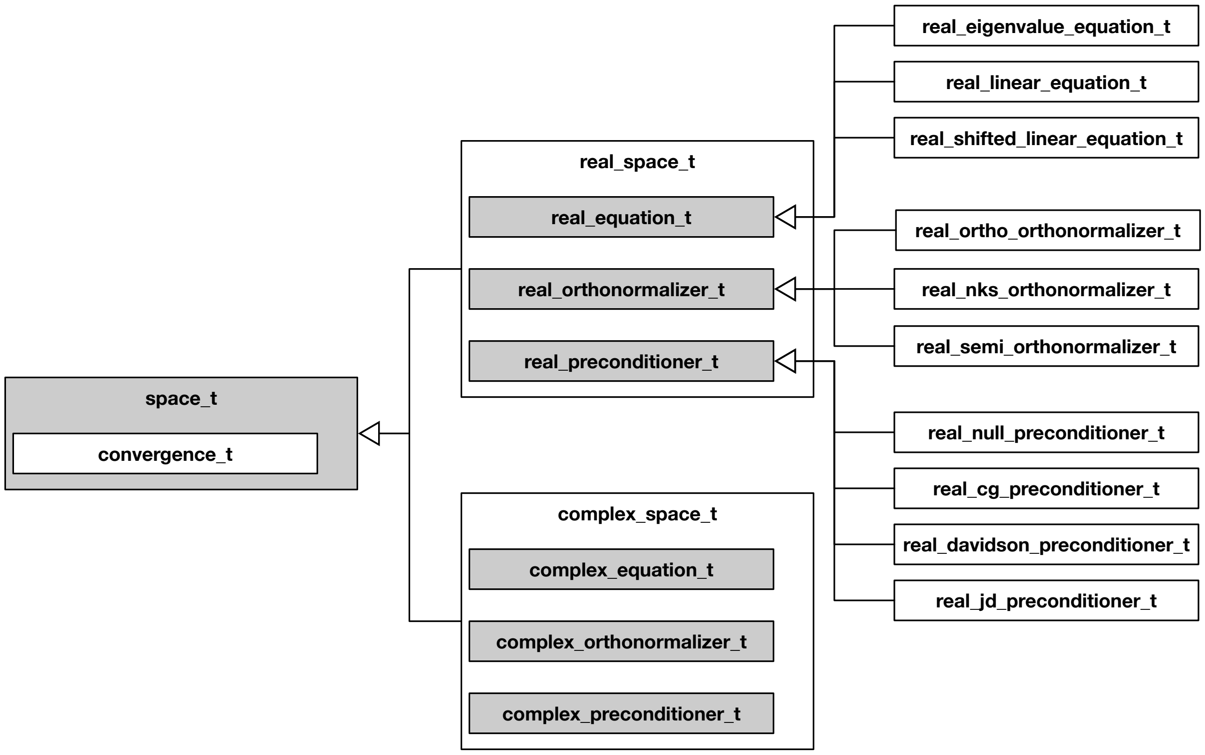

real_eigenvalue_equation_t
real_linear_equation_t
real_shifted_linear_equation_t
real_space_t
real_equation_t
real_ortho_orthonormalizer_t
real_orthonormalizer_t
real_nks_orthonormalizer_t
real_semi_orthonormalizer_t
real_preconditioner_t
space_t
convergence_t
real_null_preconditioner_t
real_cg_preconditioner_t
complex_space_t
complex_equation_t
real_davidson_preconditioner_t
real_jd_preconditioner_t
complex_orthonormalizer_t
complex_preconditioner_t

Figure (3) Simplified class diagram of libkrylov components. Abstract classes are shown in gray, concrete classes in white. Inheritance relationships are indicated by arrows.

```
program use_krylov
    integer :: err, ind
    real :: v(4, 1), s(4, 1), res
    err = krylov_initialize()
    ind =  krylov_add_space('r', 's', 'e', 4, 1, 1)
    v = reshape([1.0, 0.0, 0.0, 0.0], [4, 1])
    err = krylov_set_real_space_vectors(ind, 4, 1, v)
    err = krylov_solve_real_equation(ind, multiply)
    if (err /= 0) stop 1
    err =  krylov_get_real_space_solutions(ind, 4, 1, s)
    res = krylov_get_space_last_residual_norm(ind)
    err = krylov_finalize()
contains
    function multiply(n, m, v, p)
        integer :: n, m
        real :: v(n, m), p(n, m), m(4, 4)
        mat = reshape([5.0, 4.0, 1.0, 1.0, 4.0, 5.0, 1.0, 1.0, &
                       1.0, 1.0, 4.0, 2.0, 1.0, 1.0, 2.0, 4.0], [4, 4])
        p = matmul(mat, v)
        multiply = 0
    end function multiply
end program use_krylov
```

Figure (4) Example of calling libkrylov from Fortran. Program variables: `n`: dimension of coefficient matrix, `m`: current subspace dimension, `t`: number of solutions, `v(n, m)`: basis vectors, `p(n, m)`: matrix–vector products, `s(n, t)`: solutions, `res`: residual norm, `ind`: space index, `err`: error code. An explicit coefficient matrix `mat` is used for simplicity. In real implementations, the matrix–vector products are formed "on the fly".

```
#include "ckrylov.h"
int main() {
    long err, ind;
    double v[4] = {1.0, 0.0, 0.0, 0.0}, s[4], res;
    err = ckrylov_initialize();
    ind = ckrylov_add_space("r", 1, "s", 1, "e", 1, 4, 1, 1);
    err = ckrylov_set_real_space_vectors(ind, 4, 1, v);
    err = ckrylov_solve_real_equation(ind, multiply);
    if (err != CKRYLOV_OK) exit(1);
    err = ckrylov_get_real_space_solutions(ind, 1, s);
    res = ckrylov_get_space_last_residual_norm(ind);
    err = ckrylov_finalize();
    exit(0);
}
int multiply(const long *n, const long *m, const double *v, double *p) {
    double mat[] = {5.0, 4.0, 1.0, 1.0, 4.0, 5.0, 1.0, 1.0,
                    1.0, 1.0, 4.0, 2.0, 1.0, 1.0, 2.0, 4.0};
    for (long i = 0; i < *n * *m; ++i) p[i] = 0.0
    for (long i = 0; i < *m; ++i) {
        for (long j = 0; i < *n; ++j) {
            for (long k = 0; k < *n; ++k) {
                p[k + *m * i] += mat[k + *n * j] * v[j + *n * i];
            }
        }
    }
    return CKRYLOV_OK;
}
```

Figure (5) Example of calling libkrylov from C. See Fig. 4 for variable definitions. Integer constant `CKRYLOV_OK` indicates successful exit.

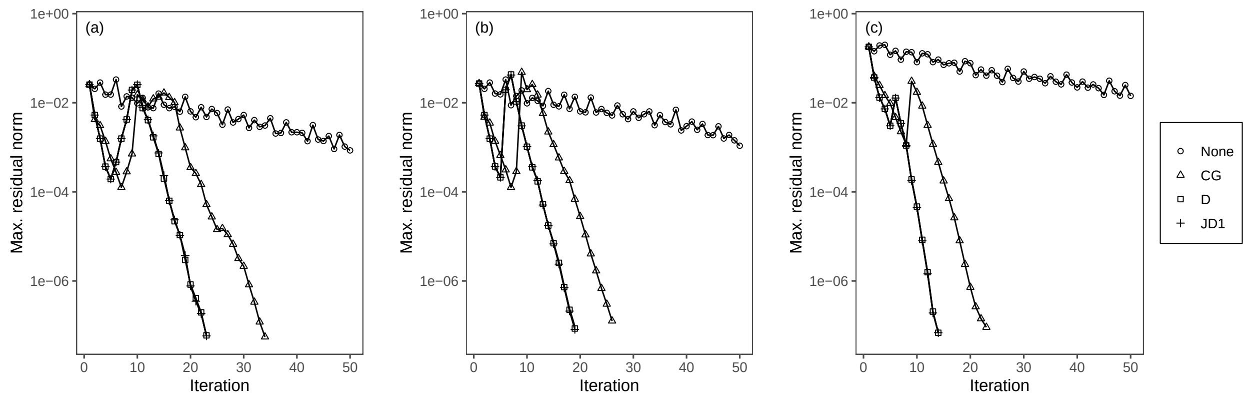


Figure (6) Convergence of the maximum residual norms of the orthonormal Krylov subspace algorithm for calculations of the p lowest electronic excitations of trans-thioindigo without preconditioning (None) and with diagonal (conjugate gradient, CG), Davidson (D), and Jacobi–Davidson preconditioner, variant 1 (JD1). (a) $p = 1$, (b) $p = 2$, (c) $p = 10$. $q_0 = p$ basis vectors are used as starting subspace basis.

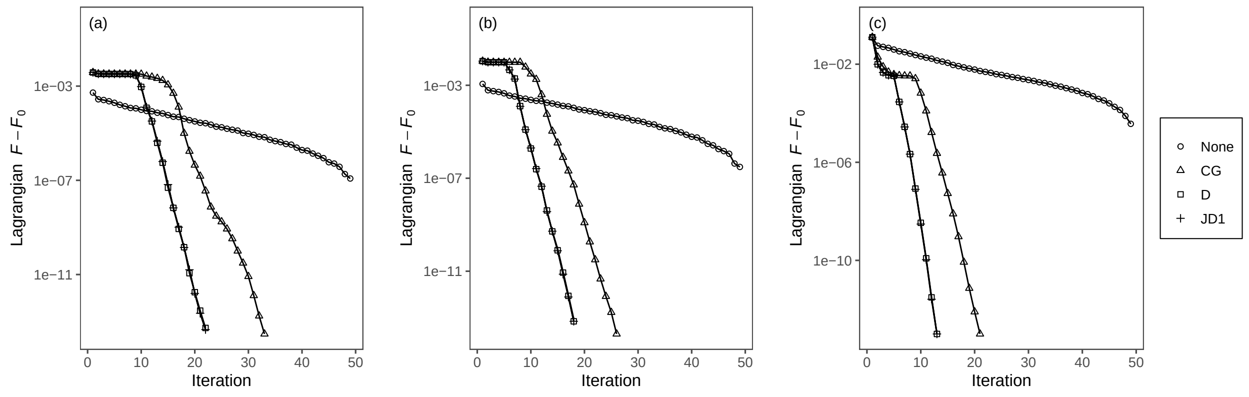


Figure (7) Convergence of the Lagrangian functional F for the p lowest electronic excitations of trans-thioindigo without preconditioning (None) and with diagonal (conjugate gradient, CG), Davidson (D), and Jacobi–Davidson preconditioner, variant 1 (JD1). (a) $p = 1$, (b) $p = 2$, (c) $p = 10$. F_0 is the stationary value. $q_0 = p$ basis vectors are used as starting subspace basis.

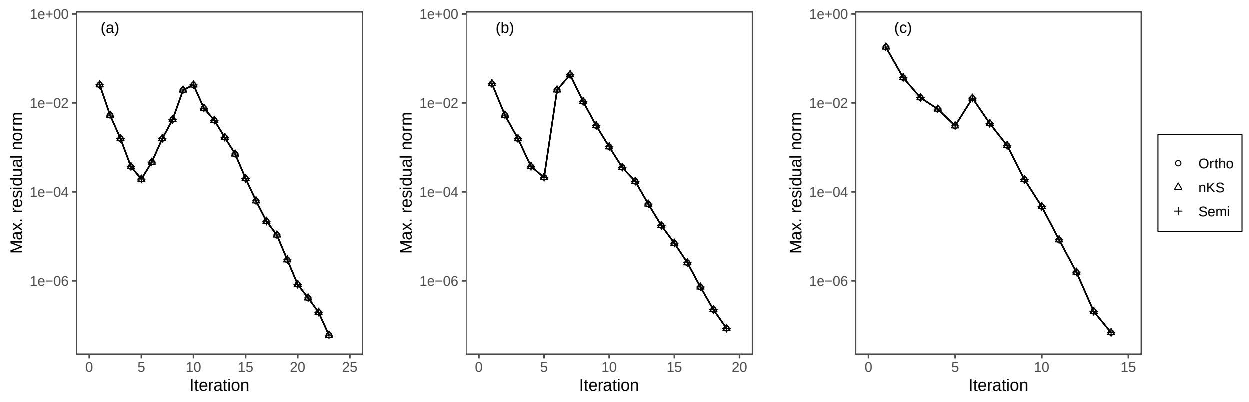


Figure (8) Convergence of the maximum residual norms of the Krylov subspace algorithm for the p lowest electronic excitations in trans-thioindigo with Davidson preconditioner using orthonormal algorithm (Ortho), nonorthonormal Krylov subspace method (nKs), and semiorthonormal method (Semi). (a) $p = 1$, (b) $p = 2$, (c) $p = 10$. $q_0 = p$ basis vectors are used as starting subspace basis.

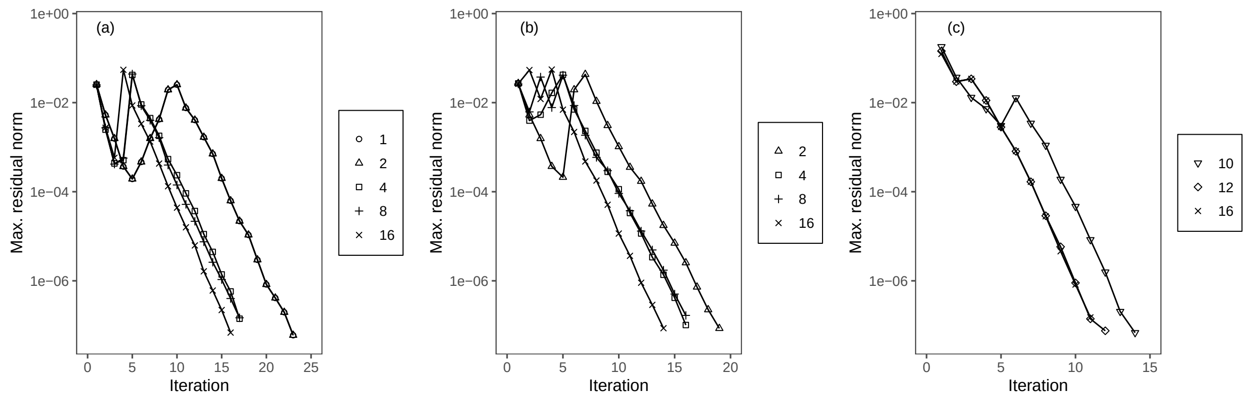


Figure (9) Convergence of the maximum residual norms of the orthonormal Krylov subspace algorithm for the p lowest electronic excitations in trans-thioindigo with different starting subspace basis dimensions q_0 using Davidson preconditioner. (a) $p = 1$, $q_0 = 1, 2, 4, 8, 16$, (b) $p = 2$, $q_0 = 2, 4, 8, 16$, (c) $p = 10$, $q_0 = 10, 12, 16$.

	Equation	**Projected Equation**	**Residuals**
Eigenvalue problem	$\mathbf{AX} = \mathbf{\Omega X}$	$\mathbf{a}^{(k)}\mathbf{x}^{(k)} = \mathbf{s}^{(k)}\mathbf{x}^{(k)}\mathbf{\Omega}^{(k)}$	$\mathbf{R}^{(k)} = \mathbf{AX}^{(k)} - \mathbf{X}^{(k)}\mathbf{\Omega}^{(k)}$
	$\mathbf{X}^\dagger\mathbf{X} = \mathbf{1}$	$\mathbf{x}^{(k)\dagger}\mathbf{s}^{(k)}\mathbf{x}^{(k)} = \mathbf{1}$	
Linear equation	$\mathbf{AX} = \mathbf{P}$	$\mathbf{a}^{(k)}\mathbf{x}^{(k)} = \mathbf{p}^{(k)}$	$\mathbf{R}^{(k)} = \mathbf{AX}^{(k)} - \mathbf{P}$
Shifted linear equation	$\mathbf{AX} - \mathbf{X}\boldsymbol{\omega} = \mathbf{P}$	$\mathbf{a}^{(k)}\mathbf{x}^{(k)} - \mathbf{s}^{(k)}\mathbf{x}^{(k)}\boldsymbol{\omega} = \mathbf{p}^{(k)}$	$\mathbf{R}^{(k)} = \mathbf{AX}^{(k)} - \mathbf{X}^{(k)}\boldsymbol{\omega} - \mathbf{P}$

Table (1) Linear problems solved by libkrylov, the corresponding projected equations on Krylov subspace $\mathcal{K}^{(k)}$, and definitions of the residual matrices. See text for definitions.

Compound	Basis	n	p	Coefficient matrix $\mathbf{A}$				Iteration count K			
				κ_2^{-1}	ρ_p	$\overline{\delta\Omega}_p$	$\overline{\Omega}_p$	CG	D	JD1	JD2
Na_8	SVP	3344	10	$1.13 \cdot 10^{-3}$	1.060	0.012	0.053	13	9	9	9
	TZVP	9328	10	$9.85 \cdot 10^{-4}$	1.040	0.012	0.051	15	11	11	11
$(H_2O)_6$	SVP	3420	10	$1.01 \cdot 10^{-2}$	1.005	0.002	0.261	22	10	10	10
	SVPD	6120	10	$9.32 \cdot 10^{-3}$	1.009	0.002	0.243	20	8	8	8
	TZVP	6840	10	$3.52 \cdot 10^{-3}$	1.018	0.001	0.254	21	8	8	8
S_8	SVP	5120	10	$1.41 \cdot 10^{-3}$	1.000	0.006	0.133	38	19	19	19
	SVPD	9728	10	$1.37 \cdot 10^{-3}$	1.000	0.006	0.129	38	18	18	18
	TZVP	14848	10	$1.16 \cdot 10^{-3}$	1.000	0.006	0.131	41	21	21	21
$B_{10}C_2H_{12}$	SVP	7067	10	$1.97 \cdot 10^{-2}$	1.009	0.006	0.265	34	16	15	15
	SVPD	11063	10	$1.95 \cdot 10^{-2}$	1.005	0.004	0.263	29	14	14	14
	TZVP	15059	10	$7.68 \cdot 10^{-3}$	1.008	0.004	0.263	29	13	13	13
Ag_6	SVP	7353	10	$1.58 \cdot 10^{-3}$	1.024	0.012	0.116	34	14	14	14
	TZVP	10431	10	$7.89 \cdot 10^{-4}$	1.024	0.012	0.115	33	14	17	18
C_{20}	SVP	13200	1	$2.83 \cdot 10^{-3}$	1.027	0.009	0.037	15	12	12	12
			2		1.000	0.010	0.037	11	11	11	11
			10		1.052	0.021	0.076	36	23	24	24

Compound	Basis	n	p	Coefficient matrix $\mathbf{A}$				Iteration count K			
				κ_2^{-1}	ρ_p	$\overline{\delta\Omega}_p$	$\overline{\Omega}_p$	CG	D	JD1	JD2
Coumarin	SVP	5548	10	$5.86 \cdot 10^{-3}$	1.033	0.022	0.187	22	14	14	14
	SVPD	8968	10	$6.02 \cdot 10^{-3}$	1.005	0.019	0.185	23	16	16	16
	TZVP	12882	10	$2.12 \cdot 10^{-3}$	1.043	0.022	0.185	21	13	14	14
DMABN	SVP	6435	10	$8.65 \cdot 10^{-3}$	1.003	0.012	0.206	24	13	13	13
	SVPD	10179	10	$5.41 \cdot 10^{-3}$	1.042	0.011	0.162	25	14	14	14
	TZVP	14118	10	$3.14 \cdot 10^{-3}$	1.020	0.012	0.199	25	14	14	14
YP	SVP	7095	10	$5.67 \cdot 10^{-3}$	1.021	0.015	0.179	19	11	11	11
	SVPD	11610	10	$5.79 \cdot 10^{-3}$	1.010	0.013	0.175	21	12	12	12
DPA	SVP	8640	10	$8.15 \cdot 10^{-3}$	1.004	0.027	0.183	22	12	12	12
Anthracene	SVP	9353	10	$8.63 \cdot 10^{-3}$	1.009	0.023	0.172	32	16	16	16
	SVPD	14711	10	$8.46 \cdot 10^{-3}$	1.004	0.018	0.169	31	16	15	15
Luciferin	SVP	16416	10	$1.32 \cdot 10^{-3}$	1.031	0.013	0.148	22	12	12	12
cis-Thioindigo	SVP	19152	10	$7.27 \cdot 10^{-4}$	1.002	0.011	0.111	25	15	15	15
trans-Thioindigo	SVP	19152	1	$8.99 \cdot 10^{-4}$	1.040	0.004	0.082	34	23	23	23
			2		1.011	0.005	0.084	27	19	19	19
			10		1.024	0.013	0.112	23	14	14	14
Crystal violet	SVP	44200	10	$5.31 \cdot 10^{-3}$	1.045	0.023	0.123	22	15	14	14

Table (2) Characteristics of the coefficient matrix $\mathbf{A}$ (of size n, in a.u.) and iteration counts K of orthonormal Krylov subspace algorithms for computing the p lowest electronic excitations using TDDFT in the TDA approximation with diagonal (conjugate gradient, CG), Davidson (D), and Jacobi–Davidson preconditioners, variants 1 (JD1) and 2 (JD2). κ_2^{-1} is the inverse 2-norm condition number of $\mathbf{A}$, $\overline{\delta\Omega}_p$ is the average eigenvalue shift, and ρ_p is the $(p+1)/p$ eigenvalue ratio. The number of starting vectors was $q_0 = p$, convergence threshold was $\tau = 10^{-7}$. See text for details.