D 3 Numerical methods for the eigenvalue problem in electronic structure computations [1]

Edoardo Di Napoli
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH

Contents

[1] Lecture Notes of the 45^{th} IFF Spring School "Computing Solids - Models, ab initio methods and supercomputing" (Forschungszentrum Jülich, 2014).

1 Introduction

Every Density Functional Theory (DFT) method is grounded on a variational principle directly inspired by the fundamental theorem of Kohn and Hohenberg [1], and its practical realization [2]. Central to DFT is the solution of a large number of coupled one-particle Schrödinger-like equations known as Kohn-Sham (KS)

$$\left(\frac{\hbar^2}{2m}\nabla^2 + \mathcal{V}_{\text{eff}}[n(\mathbf{r})]\right)\phi_i(\mathbf{r}) = E_i\phi_i(\mathbf{r}) \quad ; \quad n(\mathbf{r}) = \sum_i f_i\phi_i(\mathbf{r}).$$

Due to the dependence of the effective potential $\mathcal{V}_{\text{eff}}$ on the charge density $n(\mathbf{r})$, in itself a function of the orbital wave functions $\phi_i(\mathbf{r})$, the KS equations are non-linear and are generally solved by a sequence of self-consistent field (SCF) cycles.
The KS equations need to be "discretized" in order to be solved numerically. Intended in its broadest numerical sense, the discretization translates the KS equations in a non-linear eigenvalue problem. Eigenproblems generated by distinct discretization schemes have numerical properties that are often substantially different; for sake of simplicity we can group most of the schemes in three classes. The first and the second classes expand each of the one-particle orbital wave functions $\phi_i(\mathbf{r})$ appearing in the KS equations on a specific set of basis functions

$$\phi_i(\mathbf{r}) \longrightarrow \phi_{\mathbf{h},i}(\mathbf{r}) = \sum_{\mathbf{G}} c^{\mathbf{G}}_{\mathbf{h},i}\psi_{\mathbf{G}}(\mathbf{h},\mathbf{r}), \tag{1}$$

where $\mathbf{G}$ and $\mathbf{h}$ represent cumulative general sets of indices.
The first class makes use of simple plane waves $\psi_{\mathbf{G}}(\mathbf{k},\mathbf{r}) \sim e^{i(\mathbf{k}+\mathbf{G})\cdot\mathbf{r}}$ with **G** and **k** (here the more commonly used **k** replaced **h** index) being vectors in momentum space. This basis set is very simple to handle in Fourier space where the kinetic energy operator $\frac{\hbar^2}{2m}\nabla^2$ is diagonal. On the contrary, the potential term $\mathcal{V}_{\text{eff}}[n(\mathbf{r})]$ gives rise to a large number of off-diagonal terms. Moreover, when close to the origin of the atomic Coulomb potential, plane waves can oscillate quite wildly giving rise to computational difficulties. For the latter reason plane waves are usually utilized in combination with pseudo-potentials where the singular part of the Coulomb term is replaced with a softer function emulating the screening effects of the core electrons. The resulting pseudo-potential contains both local and non-local terms inducing dense eigenproblems.
The second class resorts to localized functions $\psi_{\mathbf{G}}(\mathbf{h},\mathbf{r}) \sim R_\ell(\mathbf{r}_a)Y_{\ell,m}(\hat{\mathbf{r}}_a)$ which combine radial functions around an atom with spherical harmonics. One popular example are the Gaussian type orbitals (GTO)[3, 4]. In this basis set the radial functions are equal to $R_\ell(\mathbf{r}_a) = r_a^\ell e^{-\alpha_p r^2}$, where $\mathbf{G} = (a,\ p)$ is indexing the atom-localized primitive Gaussians, **h** parametrizes quantum numbers ℓ and m, and $c^{\mathbf{G}}_{\ell,m,i}$ include the contraction coefficients and normalization constants. The GTOs set is particularly convenient since it capitalizes on the fact that integrals of multiple products of Gaussians can be easily reduced, in a chain of simplifications, to products of single Gaussian integrals. On the opposite many functions are needed to represent faithfully each electronic orbital, ending in an expensive bookkeeping process. The net result is that methods based on GTOs, as well as other kind of localized orbitals, are usually quite accurate but require a big deal of optimization to ensure the basis set is complete.
A special place is occupied by methods based on Linearized Augmented Plane Waves (LAPW) [5, 6], where a mix of radial functions and plane waves are used. The main advantage of these methods reside in the ability of employing the full potential without the need of distinguishing

between core and valence electrons. For this reason this basis set is usually considered quite accurate – even if expensive – for the simulation of transition metals. Since they give rise to dense problems, for the purpose of our classification, we just include LAPW-based methods in the first class.
Methods in the third class do not use an explicit basis set but discretize the KS equations using functions centered on a uniform mesh in real space. The easiest real-space methods use high-order finite differences [7] but there are also implementation using finite elements or wavelets [8, 9]. In the latter case, it is possible to have an adaptive mesh closer to the nucleus using the scaling properties of wavelets. These methods use functions that are localized, a technique which allows the development of order n methods. In this formalism the potential entries in the Hamiltonian matrix decay quite radiply away from the main diagonal while the size of the matrix is proportional with the total number of grid points leading to quite large eigenproblems.
Eigenvalue problems emerging from the first two discretization classes consist of **dense** matrices of **small-to-moderate** size while, within real space methods, one ends up with **very large** and **sparse** matrices. While for most DFT methods only a **fraction** of the eigenspectrum is required, the magnitude of such fraction can vary wildly from method to method (even within the same class). In addition, depending on the choice of basis set, the eigenproblems could be either standard or generalized. In the latter case the numerical properties of the overlap matrix strongly depend on the over-completeness of the basis set. In some cases this amounts to deal with eigenproblems with large **condition** number.
Due to the dramatically different set of properties of the eigenproblems, each DFT method uses a distinct strategy in solving for the required eigenpairs. For instance it is quite common that methods ending up with dense problems and a fraction of the eigenspectrum larger than 1% use direct eigensolvers. On the opposite many real space methods end up with very sparse and quite large matrices which are not even generated explicitly. Consequently these methods make use of iterative eigensolver based on Krylov- or Davidson-like subspace constructions. From the point of view of software packages for distributed memory architectures, the choice between direct or iterative eigensolvers leads to the use of traditional parallel libraries like, respectively, ScaLAPACK [10] or PARPACK [11]. Not satisfied with traditional libraries, some of the newer implementations ended up developing their own tailored eigensolver (e.g. [12, 13]).
In the following section we will address both direct and iterative eigensolvers, their properties, general parallel implementation strategies as well as some tailored algorithms. Due to the vastness of the subject we will not be exhaustive but will use specific algorithmic examples to illustrate the differences in their computational approach. The careful reader can expect to gain some general insight on which algorithm is best suited to which DFT implementation.

2 Definitions and tools

Let us first introduce some definitions and terminology which are common among specialists and will be used in the rest of the chapter. We define three distinct type of eigenproblems. They are all identified by the equation

$$Ax = \lambda Bx \tag{2}$$

where $A,\ B \in \mathbb{C}^{n\times n}$ are given matrices[2] and one seeks the unknown scalars $\lambda \in \mathbb{R}$ and the associated vectors $x \in \mathbb{C}^n$. The latter are referred to as, respectively, *eigenvalues* and

[2]In this chapter we always use the letter n to indicate the size of the matrices

eigenvectors and are usually displayed as an *eigenpair* $(\lambda,\ x)$. It is also common practice to designate the pair $(A,\ B)$ as an *eigenpencil*.
In the most general case both $A,\ B$ are Hermitian symmetric ($A^\dagger = A,\ B^\dagger = B$) and B is positive or negative definite[3]. In this case we will refer to Eq. (2) as the *generalized Hermitian eigenvalue problem*, GHEVP in brief . If $B = I$ then Eq. (2) reduces to the *standard Hermitian eigenvalue problem* (HEVP) $Ax = \lambda x$. In the particular case where A is also tridiagonal ($a_{ij} = 0\ \ j > i+1$) the eigenproblem is identified as *symmetric tridiagonal eigenvalue problem* (STEVP). In the latter case all non-zero entries of the matrix A are real-valued.
It is important that B is a definite matrix for the GHEVP to be well-posed. As we already mentioned in the Introduction a measure of such property is the *condition number*. This number is defined as

$$\kappa(B) \doteq \frac{\|B\|_2}{\|\ B^{-1}\ \|_2} = \frac{\max_i(\sigma_i)}{\min_j(\sigma_j)} = \frac{|\lambda_{\max}(B)|}{|\lambda_{\min}(B)|}.$$

It can be expressed in relation to the singular value decomposition of $B = W\Sigma V$ where $W,\ V$ are two unitary matrices and $\Sigma = \mathrm{diag}(\sigma_1 \dots \sigma_n)$ is a diagonal matrix holding the singular values of B. In the case of GHEVP such values corresponds just to the absolute value of the largest and smallest eigenvalues of B. Consequently a large condition number implies that B has one or more eigenvalues very close to zero. When this happens some of the eigenvectors of the GHEVP are close to linearly dependent and B can be hardly inverted or factorized. Consequently it becomes very difficult to solve the eigenproblem.

Metrics and performance — The general concept of performance can be associated with several metrics. One could consider the total *CPU time* to completion as the measure of the performance of an algorithm implementing an algebraic operation. In many cases this is considered an important performance metric to compare two algorithmic variants of the same operation tested in exactly the same condition.
Alternatively one could instead decide that the best algorithm is the one performing the least number of elementary operations. One refers to the number of elementary operations of a algebraic transformation as its *complexity*. For example the complexity of a real-valued matrix-vector product is $2n^2-n$ (n multiplications and $n-1$ additions per vector entry). It is customary to consider only the leading term in n contributing to the complexity. For the matrix-vector product the complexity would then be indicated as $\mathcal{O}(n^2)$. The complexity of a matrix operation is directly correlated with the number of *floating point operations* (flop) a computer is capable to handle.
The number of flop is a more fundamental quantity with respect to the complexity of an algorithm and it is at the base of the modern way of judging the efficiency of a numerical computation. This is due to the fact that some elementary operations costs more (in terms of flop counts) than others. In this chapter, as it is done in current literature, we adopt as definition of *performance* the flop count rate which is defined as

$$\text{PERFORMANCE}\ (Flops) \doteq \frac{\text{NUMBER OF OPERATIONS}}{\text{TIME}\ (sec)}.$$

[3]It is common practice in numerical linear algebra to represent the hermitian conjugation with the letter H (or T in the case of real symmetric matrices). Contrary to this habit we use in this chapter the symbol †, commonly used among physicists.

Since it is important to put in perspective the absolute performance with the computing architecture used, we also introduce the *theoretical peak* performance of a machine defined as

$$\begin{array}{c}\text{THEORETICAL}\\ \text{PEAK PERFORMANCE}\end{array} \doteq (\text{FREQ.}) \cdot (\#\ \text{OF CORES}) \cdot \frac{\#\ \text{OF ELEM. OPERATIONS}}{\text{CYCLE}}.$$

With this definition in hand a better measure and a useful tool is the *efficiency* η of a routine, defined as its performance over the theoretical peak performance. For example one may choose an expensive (in terms of flop counts) but efficient algorithmic variant respect to a cheaper but less efficient one. If the efficiency of the former can compensate for the higher complexity, the first routine will be faster in terms of CPU time. Clearly this is not the only consideration that should guide the final user in choosing an algorithm.
Another important metric to consider is the amount of memory necessary to perform a specific computation. The *memory requirements* of an algorithm – also known as *workspace* – is the amount of main cache memory which is needed to complete the computations of a certain algorithm. For example one may want to choose a more "green" computational approach and minimize the movement of data between memory and processors[4] at the cost of some performance. In fact memory has became an expensive commodity both in terms of memory size available per core and consumption of energy. Moreover some algorithms may require a large chunk of cache memory reserved for workspace: if a specific architecture have limited memory per core such algorithms may be bound by the size of the input data they can handle.
Last but not the list we want to mention *accuracy* among the possible metrics which are used to analyze the performance of an algorithm implementation. Some algorithms may, in fact, compromise on accuracy in order to improve the performance. Moreover some algorithms are intrinsically more stable and accurate than others; a property that can play an important role in choosing the correct algorithm for a specific application. When dealing with eigensolvers, accuracy of the results is measured by the value of the *residual norm* and *numerical orthogonality* defined, for the HEVP, as

$$r(\lambda, x) \doteq \max_i \frac{\parallel Ax_i - \lambda x_i \parallel}{\parallel A \parallel n\epsilon} \qquad ; \qquad O(x) \doteq \max_{i \neq j} \frac{\|x_j^\dagger x_i\|}{n\epsilon}.$$

Here ϵ indicates the relative machine precision. The above definition can be generalized in an obvious manner to all the other eigenproblems.

Parallelism and scalability — When a program is run on more than one processor, the metrics above are not enough to describe the "quality" of an algorithm and its implementation in a routine. What is lacking are tools addressing the ability of the algorithm to run efficiently in a parallel fashion. The *speed-up* of a routine provides a first tool in this direction. Speed-up can be defined in several ways and we refer to the works of Amdahl and Gustavsson for a theoretical oriented discussion [14, 15]. For our practical purposes we define below the speed-up for *strong scalability* and *weak scalability*

$$\text{SPEED-UP}_{strong} = \zeta_s \doteq \frac{t_{ref}(n)}{t_p(n)} \qquad ; \qquad \text{SPEED-UP}_{weak} = \zeta_w \doteq \frac{t_{ref}(n)}{t_{\beta p}(\alpha(n))}. \tag{3}$$

In the above equations, $t_{ref}(n)$ and $t_p(n)$ indicate the execution time measured in seconds for a reference hardware (e.g. one core) and p processors respectively. Strong scalability measures

[4]Data movement is by far the most energy expensive process in a computation

the performance of an algorithm when the number of processors is increased keeping fixed the size of the data (so the flop count). A perfectly scalable algorithm then should have its execution halved each time the number of processors in doubled.
Almost every routine contains portions that are intrinsically serial and consequently will never scale. For this reason a better measure of the effective scalability of an algorithm is its ability to process larger sets of data (in our case eigenproblems of increasing size) as more processors are available. Weak scalability parametrizes this property by showing the speed-up an algorithm achieves when increasing the size of the system and proportionally augmenting the number of processors. What is kept constant is the flop count per processor. As a consequence the constant β is a function of α. For example if the complexity of an algorithm is $\mathcal{O}(n^3)$ every time n is doubled the number of processor has to be increased by a factor of $\beta = 2^3$.
Similarly to the performance efficiency η, we can introduce the *parallel efficiency* for both strong and weak scalability

$$e_s \doteq \frac{t_{ref} \cdot p_{ref}}{t_p \cdot p} \qquad ; \qquad e_w \doteq \frac{t_{ref} \cdot p_{ref} \cdot \alpha_{ref}(n)}{t_p \cdot p \cdot \alpha_p(n)}$$

As for η these definitions help normalizing the scalability of the routine under scrutiny. In other words perfect scalability corresponds to an horizontal line in correspondence of the dimensionless value 1.
Two important concepts for codes which run on parallel architectures are the *algorithmic block sizes* and the *distribution block sizes*. In the first case one refers to the sizes $m_b \times n_b$ of the blocks $A_{s,t}$ a matrix A of arbitrary data type is partitioned in

$$A = \begin{pmatrix} A_{0,0} & \dots & A_{0,N-1} \\ \vdots & & \vdots \\ A_{M-1,0} & \dots & A_{M-1,N-1} \end{pmatrix},$$

with the exception of boundaries blocks which can be smaller. Similarly the total number of p processes involved in the computation are logically viewed as a two-dimensional cartesian grid having distribution block sizes r and c with $p = r \times c$. Each block $A_{s,t}$ is is distributed over the grid in such a way that the process (s, t) owns, in a contiguous manner, the blocks

$$A = \begin{pmatrix} A_{\gamma,\delta} & A_{\gamma,\delta+c} & \dots \\ A_{\gamma+r,\delta} & A_{\gamma+r,\delta+c} & \dots \\ \vdots & \vdots & \end{pmatrix},$$

where $\gamma \equiv (s + \sigma_r) \mod r$ and $\delta \equiv (t + \sigma_c) \mod c$, and σ_r and σ_c are arbitrarily chosen alignment parameters. We will see in later sections how both these concepts are crucial for high-performance computing.

3 Direct eigensolvers

Eigensolvers are categorized by the choice of approach that goes from the input matrices A and B, defining the eigenproblem, to the eigenpairs $(\lambda,\ x)$ characterizing its solution. For direct eigensolvers this path goes through the *direct* diagonalization of the eigenproblem matrices.

In other words, as part of the solution process, each matrix defining the eigenproblem is algebraically manipulated so as to bring it to diagonal form. In such form its diagonal elements correspond to the eigenvalues of the problem whose eigevectors can be subsequently computed. The diagonalization process is usually carried on with a series of transformations which maintain the symmetry properties of the matrix and, most importantly, its spectrum. In the case of Hermitian eigenvalue problems this target is achieved with a series of similarity transformations which generically modify the values of every matrix entry. In particular if the matrix A, defining the problem $Ax = \lambda x$, has a substantial number of zero entries, the transformed matrix $\tilde{A} = SAS^\dagger$ is usually densely populated. Consequently the diagonalization process is convenient for eigenproblems with matrices that are already dense[5] while it is disadvantageous for sparse matrices. For the latter a method preserving the sparsity structure, and so limiting the total number of floating point operations to solution, is preferred.
Since direct eigensolvers act on all the entries of the eigenproblem matrices, the number of elementary operations, which are performed during the diagonalization, is directly proportional to the size n of the problem. We will see in the next sections that complexity, performance and memory workspace constraints are the most important parameters guiding the computational scientist in selecting the appropriate algorithm for its needs.

3.1 The stages of a direct eigensolver and their algorithmic variants

In Sec. 2 we have defined three distinct type of eigenproblems based on the properties of the matrices associated with them. These eigenproblems can also be seen as a chain of nested problems – GHEVP $\rightarrow$ HEVP $\rightarrow$ STEVP – where each type is connected with the previous one by a non-singular linear transformation which preserve the spectrum. Such linear transformation can be seen as the action of a pair of invertible matrices K and M on the eigenpencil matrices $(A, B) \rightarrow (KAM, KBM)$. In order to preserve the symmetry of the eigenpencil the transformation KXM needs to satisfy the additional requirement $M = K^\dagger$ which restrict us to similarity transformations.
In practice the path that goes from a GHEVP to the computation of its complete (or partial) set of eigenpairs can be schematically divided in six stages. Along the road we will recover the HEVP and the STEVP and their solutions, so that there is no need to describe these other two type of problems and their path to solution. Since we are dealing with direct solvers in each stage we will operate just on the input matrices (A, B) defining the GHEVP.

(i) This stage consists in factoring the B matrix in its Cholesky components $B = LL^\dagger$, where L is a lower triangular matrix. This factorization is unique and possible only if the matrix B is positive definite. A measure of positive-definiteness is given by the condition number $\kappa(B)$. If this has a very large value it may signal that $\lambda_{\min}(B) \approx 0$ making really hard to numerically compute the factorization. In some DFT methods this condition has to be verified in advance before attempting the factorization. This stage is referred to as *Cholesky decomposition*.

(ii) In the second stage the matrix L is used to perform the first linear transformation bringing B to diagonal form $B \rightarrow L^{-1}LL^\dagger(L^\dagger)^{-1} = I$ and $A \rightarrow L^{-1}A(L^\dagger)^{-1} = C$. Consequently the eigenpencil is now reduced to (C, I) corresponding to the standard eigenvalue problem $Cy = \lambda y$. While the eigenvalues are preserved by this transformation, the same

[5]The term *dense* is commonly used to address matrices having a number of non-zero entries greater than few percentage points.

cannot be said by the eigenvectors y, which are related to the original one by $y = L^{\dagger}x$ and will motivated the last stage. The second stage is known as *reduction to standard form*.

(iii) Evidently if one has to solve for just an HEVP the first two stages are redundant and one can start directly from this stage. At this point one builds a unitary transformation, parametrized by Q, which acts exclusively (the action on I is trivial) on $C \to QCQ^{\dagger} = T$ where T is a real-valued symmetric tridiagonal matrix. The eigenproblem has now been reduced to $Tz = \lambda z$. Like in the previous stage the spectrum is preserved while the eigenvectors go through another transformation $z = Q^{\dagger}y$. We will see later that there are several methods for building the unitary matrix Q. This stage goes under the name of *reduction to tridiagonal form* and it is usually the most expensive among all stages.

(iv) This stage is the core of the chain of transformations. While all the stages above are not data dependent, the solution of the STEVP substantially depends on the distribution of the eigenvalues and it is particularly sensitive to their clustering[6]. Several are the algorithmic choices for this stage. In this chapter we will consider the four most well-known: QR, Bisection & Inverse Iteration (BXINV), Divide & Conquer (D&C), and Multiple Relatively Robust Representations (MRRR). Each one relies on a different strategy so much so that the entire six stages eigensolver inherits its name by the tridiagonal solver used in stage (iv). While this could be common practice it is important to understand that there are also algorithmic variants for the other stages. Consequently several combinations of them are not only possible but quite different. Whatever is the tridiagonal solver of choice the output of this stage are the pairs (λ, z). For this reason stage (iv) is know as *solution of the tridiagonal eigenproblem*. It should be noted in passing that the STEVP also appear as a byproduct of some iterative eigensolvers, most notably the Lanczos method (see Sec. 4.1).

(v) Once we have the pairs (λ, z), it is just a question of tracing back the eigenvectors of STEVP to the eigenvectors of HEVP $y = Qz$ with the first of the so called *back-transformation*. If one was bound to solve just a standard eigenvalue problem, this would be the last stage of the chain.

(vi) Similarly to stage (v), this last stage is meant to compute the eigenvectors of the GHEVP by the *second back-transformation* $x = (L^{\dagger})^{-1}y$ leading to the desired output (λ, x).

Despite the level of complexity of the operations in each stage appears rather low, there are several algorithmic variants for each stage. For example the Cholesky decomposition could be realized in three main variants called respectively right-looking, left-looking and bordered algorithms. Each of these algorithms have distinct performance and memory signatures. Since we cannot cover the fine algorithmic details of all stages we will list, in the following, the most important algorithmic variants of only stages (iii) and (iv). For more details on the algorithmic choices for the rest of the stages we refer to the standard book by Golub and Van Loan [16].

Reduction to tridiagonal form — In general the unitary matrix Q computed at this stage is the composition of a series of projection operators, each one a unitary matrix in itself. The scope of the projection is to eliminate all the entries of C below and above the first lower and upper

[6] A cluster is loosely defined as a set of adjacent eigenvalues densely concentrated around one value with a relative distance substantially smaller than the other neighboring eigenvalues.

sub-diagonals respectively. The two most well-known methods are Givens and Householder transformations.
In the Givens method the matrix Q is the composition of a series of elementary rotation matrices G_{pq} having the only non-zero entries $g_{qq} = g_{pp} = \cos(\phi)$; $g_{ii} = 1$ for $i \neq p, q$, and $g_{pq} = -\bar{g}_{qp} = \rho \sin(\phi)$ with $\rho \in \mathbb{C}$, $\phi \in \mathbb{R}$. By construction G_{pq} is unitary and its action $C^{(1)} = G_{pq}\, C\, G_{pq}^{\dagger}$ can be chosen so as to zero out elements of C with a specific index r, $c_{qr}^{(1)} = c_{rq}^{(1)} = 0$ (the subscript $^{(k)}$ indicates how many elementary transformations were operated on C). For example one can zero out all elements $c_{1,q}\ \forall\ q$ with a sequence of transformations $(p,q) = (2,3), (2,4), \ldots (2,n)$. The resulting matrix $C^{(n-2)} = G_{2n} \ldots G_{2,3}\, C\, G_{23}^{\dagger} \ldots G_{2n}^{\dagger}$ has all elements of the first row and column equal to zero apart from $c_{1,1}^{(n-2)}, c_{1,2}^{(n-2)}, c_{2,1}^{(n-2)}$. Proceeding in the same way one can eliminate all other entries until all is left is just a tridiagonal matrix. Each Givens transformation $C^{(k)} \to C^{(k+1)}$ requires $4(n-r)$ multiplications and for each index r there are $(n-r)$ of them. Summing over all values of $r = 1, \ldots, n-2$ makes the complexity of the entire reduction to tridiagonal form $\sum_r 4(n-r)^2 \approx \frac{4}{3}n^3$. This method is particularly suitable for dense matrices which have some definite structure for the non-zero entries: by avoiding to act on the null elements the Givens method can avoid redundant computations.
The more commonly used Householder method builds Q out of elementary matrices

$$G_k = I - \beta u_k u_k^{\dagger}$$

which are both hermitian and unitary. The vector u_k and constant β are chosen in such way to zero out all the entries of the k column of the matrix $G_k C^{(k)}$ with row indices bigger than $k+1$. By acting on the left with the inverse of G_k (which is the same as the G_k) one eliminates also all entries of the k row with column index bigger than $k+1$. The result is similar to the chain of Givens rotations described above apart from the fact that an Householder transformation achieves this result in a more economical way. In fact the matrix G_k is never used explicitly and only rank two updates are used instead (see Golub Van Loan for details). Because of this property each elementary transformation $C^{(k)} \to C^{(k+1)} = G_k C^{(k)} G_k$ requires only $2(n-k)^2$ operations for a total of $\sum_k 2(n-r)^2 \approx \frac{2}{3}n^3$
A third alternative for the reduction to tridiagonal form was developed relatively recently by Bischof et Al. [17] and goes by the name of two-step successive band reduction (SBR). The basic strategy is to first reduce the dense matrix of the HEVP to a banded form leveraging level 3 BLAS operations (see Sec. 3.2 for a definition of BLAS) and only subsequently reduce the banded matrix to tridiagonal form. Since only the second step uses less performant level 2 and 1 BLAS routines, SBR shows better performance with respect to the classic Householder method. On the downside SBR needs a total of $\frac{4}{3}n^3$ operations for the reduction and $2n^3$ for the accumulation of the matrix Q on the fly. This implies that SBR is a very convenient method when one is interested in only the eigenvalues, since they don't need the accumulation of the similarity matrix Q, while it may be penalizing if also the eigenvectors are required.

Tridiagonal eigensolver — Four are the main algorithms that are used to solve for the tridiagonal problem. They are distinguished by the solving strategy, the memory requirements, complexity and the ability to solve for just a subset of the spectrum. In the following we provide a short description and indicate which are the most suitable for DFT computations.
QR uses a series of similarity transformations, preserving the tridiagonal structure of T, which turn progressively off the sub-diagonal elements. The algorithm achieves this target by a so

called bulge-chasing procedure which includes implicit shifts and deflation techniques. The total complexity of the algorithm is $\mathcal{O}(sn^3)$, where s indicates the median number of bulge chases per eigenvalue. The QR algorithm cannot be used to solve for subset of the eigenspectrum and so it is not particularly suitable for DFT computations.
BXINV is the first of the two algorithms which is capable of solving for a subset of the eigenpairs. The algorithm uses Sturm sequences (bisection algorithm) with a total of $\mathcal{O}(kn)$ operations for k eigenvalues. If such eigenvalues are well separated then BXINV requires another $\mathcal{O}(kn)$ to compute the relative eigenvectors (inverse iteration method). In the case the eigenvalues are grouped in tight clusters the complexity can grow up to $\mathcal{O}(n^3)$. The grow in complexity is motivated by the need of re-orthogonalize eigenvectors since numerical orthogonality is not automatically guaranteed for clustered eigenvalues. This algorithm is probably the one currently most used for DFT computations involving dense eigenproblems. As we will see later BXINV is by far outperformed by its close cousin MRRR which should be preferred to it.
D&C strategy is rather well described by its name. This algorithm decomposes T in a hierarchical tree of smaller submatrices and rank-one updates. At the bottom of the tree each child submatrix is solved using a secular equation and the process is repeated going from child to father until the tree is complete. D&C cannot compute subset of the eigenspectrum and its complexity is $\mathcal{O}(n^3)$. The complexity can often be reduced substantially by a deflation process when certain entries in the eigenvectors of the submatrices are small enough. Despite being not optimal for DFT computations, D&C can be quite performant so as to be used in DFT computations provided one discards, in the end, the part of the spectrum not required.
MRRR is the second of the algorithms capable of computing for a subset of the spectrum. This algorithm is a sophisticated variation of BXINV which avoids altogether the re-orthogonalization of the eigenvectors. Consequently MRRR complexity is approximately reduced to $\mathcal{O}(n^2)$. In practice the overall complexity depends on the clustering of the eigenvalues. Due to its low complexity and the ability to solve for a portion of the eigenspectrum without the need for costly orthogonalizations, this is the most indicated algorithm for DFT computations and should almost always be preferred to BXINV.

3.2 Libraries, performance and parallelism

Since numerical linear algebra deals with vectors and matrices the most common and important operations are included in specialized and optimized libraries. Among the most well-known are the Basic Linear Algebra Sub-routines (BLAS) and the Linear Algebra PACKage (LAPACK). One of the most important practice that can improve the performance of an algorithm-derived routine is the correct use of the kernels already present in these standard libraries.

BLAS and LAPACK — The main motivations behind the Basic Linear Algebra Subroutines were modularity, efficiency, and portability through standardization [18]. The BLAS library consists of three levels, corresponding to routines for vector-vector, matrix-vector and matrix-matrix operations [19, 20]. From a mathematical perspective, it might appear that this structure introduces unnecessary duplication: For instance, a matrix-matrix multiplication (a level 3 routine) can be expressed in terms of matrix-vector products (level 2), which in turn can be expressed in terms of inner products (level 1). The layered structure is motivated by the increased efficiency of level 2 and 3 routines on architectures with a hierarchical memory. In fact BLAS 1, 2, and 3 are capable respectively of 1/2, 2 and n/2 operation counts per number of memory accesses, giving the higher-level routines a better opportunity to amortize the costly memory

accesses with calculations. With respect to the full potential of a processor, the efficiency of BLAS 1, 2, and 3 routines is roughly 5%, 20% and 90+%, respectively. Moreover, the scalability of BLAS 1 and 2 kernels is rather limited, while that of BLAS 3 is typically close to perfect. In practice, this means that level 3 routines attain the best performance and should be preferred whenever possible. In other words BLAS succeeded in providing a portability layer between the computing architecture and both numerical libraries and simulation codes.
In addition to the reference library,[7] nowadays many implementations of BLAS exist, including hand-tuned [21, 22], automatically-tuned [23], and versions developed by processors manufacturers (Intel-MKL, IBM-ESSL, AMD-CML). More importantly BLAS kernels are heavily used in most of standard libraries (e.g. LAPACK, ScaLAPACK, etc.) implementing the linear algebra operations we have described in the six stages of the GHEVP.
The LAPACK library in its modern form already includes BLAS kernels and add, on top of that, many other routines covering almost the entire spectrum of standard linear algebra operations. For example we can find in it routines to solve triangular linear systems as well as Cholesky decomposition used respectively in stage (vi) and stage (i) of the GHEVP. This library was built in the '90s on top of other library packages developed 20 years earlier like EISPACK. In LAPACK one important section is devoted to the solution of the Symmetric Eigenvalue Problem (which includes also the complex case). Routines for the solution of all three type of eigenproblems are included allowing for several choices depending, for example, on the number of eigenpairs required as well as the storing of the matrices in memory.

Tridiagonal eigensolvers on a single core — We now illustrate how some of the tools of Section 2 can be used to analyze the performance of the tridiagonal eigensolvers described in the previous subsection. Results illustrated below were conducted on a single processor using the LAPACK eigensolver implementations. Since we deal with a specific library, we refer to each eigensolver interchangeably by its acronym or the name of its relative routine as outlined in Table 1.

Table 1: *LAPACK tridiagonal eigensolvers*

Algorithm	Routine	Workspace	Eigenpairs subsets
QR	`STEV`	Real: $2n-2$	No
BXINV	`STEVX`	Real: $8n$	Yes
D&C	`STEVD`	Real: $1+4n+n^2$	No
MRRR	`STEVR`	Real: $18n$	Yes

Experimental tests using both artificial and practical matrices show without any doubt that `STEVR` and `STEVD` are typically much faster than `STEV` and `STEVX` [24]. On average MRRR is the fastest among the four algorithms with some exceptions in special cases. These conclusions are supported by an experimental measure of the "effective" complexity of each algorithm. By comparing the CPU time to completion and counting the total number of operations performed, it is possible to infer an experimental complexity value based on the assumption that the performance of each solver does not change with the size of the problem examined.
By testing on a large set of matrices it results that the effective complexity of `STEV` is $\mathcal{O}(n^{2.9})$, very similar to the one for `STEVX`. For `STEVR` the effective complexity is $\mathcal{O}(n^{2.2})$, a sligtly

[7] Available at `http://www.netlib.org/blas/`.

higher value than the theoretical one. STEVD instead registers a lower value, $\mathcal{O}(n^{2.8})$, respect to the theoretical one. So from the operation count point of view the higher performance of MRRR is quite justified.
The results for QR and BXINV are in line with the fact that neither the bulge chasing part of QR nor the bisection and inverse iteration of BXINV use any routine from the levels 2 or 3 of BLAS. The interesting piece of data is that, while MRRR execute many less operations than the other algorithms, it does so at a higher cost. In other words the number of divisions STEVR performs is always a significant fraction of the total number of operations. Since divisions cost up to 4 times more than multiplications, the measured performance is lower than expected resulting in a higher effective complexity. STEVD experiences the opposite outcome due to the combined effect of using BLAS 3 to update the eigenvector matrix, together with the mechanism of deflation. If there is a lot of deflation STEVD performs many fewer scalar operations (slow), while if there is little deflation most of the flops are performed by calls to level 3 BLAS (fast). Overall STEVD experience a speed-up respect to its theoretical complexity.
The above example illustrate how performance for an algorithm depends on several factors including complexity, block operations[8], efficiency of specialized kernels, etc. To conclude let me also remind that the performance depends also on the hierarchy of the cache memories of the computing architecture. For example STEVD may well be the fastest routine for a specific class of eigenproblems (e.g Wilkinson matrices) but the same routine uses up to $\mathcal{O}(n^2)$ workspace so that on some specialized machines such routine may not be used for large problems anymore. In the latter case the lower workspace ($\mathcal{O}(n)$) favor STEVR the optimal algorithm of choice.

ScaLAPACK, ELPA and Elemental — Where LAPACK is the most well-known library for dense numerical linear algebra single processor routines[9], ScaLAPACK is probably the most renown library for heterogeneous computing platforms. It is a adaptation of most LAPACK routines to distributed memory architectures using the Message Passing Interface (MPI) protocol. It requires PBLAS, the parallel version of BLAS, which is currently included in the library. A specific section of this library is devoted to the solution of dense eigenproblems. ScaLAPACK was also the first library using block cyclic data distribution for dense matrices together with block-partitioned algorithms. This was an important design feature introducing a tension between having block sizes that are large enough for local BLAS efficiency yet small enough to avoid inefficiency due to load-imbalance. In ScaLAPACK this tension is intrinsically included in the design by linking the distribution block sizes to the algorithmic block sizes. The library is written in Fortran and it is structured in low-level modular routines which follow the same pattern of the LAPACK ones.
As can be seen from table 2 ScaLAPACK includes only one routine for GHEVP which uses BXINV for the tridiagonal solver while all four algorithms are represented for the HEVP. This is actually not a limitation of ScaLAPACK since one can combine routines for the other stages other than (iv) and build one's own GHEVP solver. For example it was noted in [25] that one should always use the PZHENTRD for the reduction to tridiagonal form (for a square grid of processes) instead of PZHETRD: the former, which is designed for a rectangular grid of processes, is so much less performant than it is always convenient to use less cores arranged on a square grid.
The are modern alternatives to ScaLAPACK routines for dense eigensolvers. In particular in re-

[8] conceptually similar to algorithmic block size used in parallel libraries
[9] Recently some LAPACK libraries like MKL provide also multi-threaded functionalities

Table 2: *ScaLAPACK eigensolvers routines*

Algorithm	Routine (Complex)	Routine (Real)	Eigenproblem type
BXINV	PZHEGVX	PDSYGVX	GHEVP
QR	PZHEEV	PDSYEVX	HEVP
BXINV	PZHEEV	PDSYEV	HEVP
D&C	PZHEEVD	PDSYEVD	HEVP
MRRR	PZHEEVR	PDSYEVR	HEVP
D&C	–	PDSTEVD	STEVP
MRRR	–	PDSTEVR	STEVP
BXINV	–	PDSTEBZ	STEVP eigenvalues only
BXINV	PZSTEIN	PDSTEIN	STEVP eigenvectors only

cent years there has been quite a large interest in exploring new framework for parallel libraries for large dense eigenproblems. Among the several attempts in this direction the two most recent and performant are ELPA and Elemental. While ELPA is a set of routines specifically designed for eigenvalue problems, Elemental is a new complete framework for dense linear algebra operations which also include a driver for dense eigenproblems.

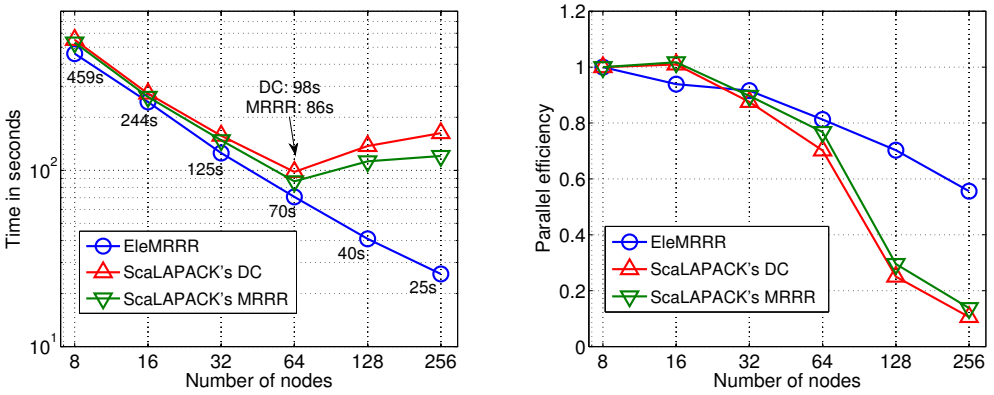
ELPA is a set of Fortran subroutines (modules) which can be compiled with an application of choice or as separate library which can also be linked to from C, or C++ code [26]. ELPA is based on the ScaLAPACK framework and cannot be used independently from it. From this point of view the ELPA library builds on top of ScaLAPACK and it is not, strictly speaking, an alternative to it. As such, necessary prerequisite libraries for ELPA include BLAS, LAPACK, Basic linear algebra communication subroutines (BLACS), and obviously ScaLAPACK. ELPA is an MPI only implementation; there are no hybrid parallelization (MPI/OpenMP) available. Consequently the ELPA library works both on a single-node, shared memory environment, as well as on large clusters of distributed memory nodes.

The library provides modular routines for two different approaches to solve large eigenvalue problems: the ELPA 1 STAGE modules, including the routine solve_evp_real , focus on reducing communication overheads and maximizing cache performance for existing standard ScaLAPACK routines (PDSYEVR). ELPA 2 STAGE routines such as solve_evp_real_2stage add extra steps into the traditional three-step HEVP approach outlined in the previous section. An intermediate banded representation of the matrix is formed during the reduction to tridiagonal form using a variant of the SBR algorithm. Likewise an intermediate banded form is generated during the back transformation stage of the calculation. The 2 Stage routines provide the real novelty of the ELPA library. Both ELPA solvers can be directed to calculate a subset of eigenpairs if preferred by the user.

Elemental is a modern framework for distributed memory dense linear algebra [27] which is designed to overcome the possible problem deriving from linking the algorithmic block size to the distribution block sizes. Elemental restricts the algorithmic block sizes to $m_b = n_b = 1$. The core of the library is the two-dimensional cyclic element-wise ("elemental" or "torus-wrap") matrix distribution. This approach is based on the observation [21] that the optimal algorithmic block size should be related to the square root of the size of the L2 cache memory. The Elemental approach for the distribution grid avoid linking the distribution with the algorithmic block sizes and so eliminates the issue between load balancing and filling the L2 cache. For a given

number $p > 1$ of processors there are several possible choices for r and c forming different grid shapes $(r, c) \doteq r \times c$. Since the grid shape can have a significant impact on the overall performance, careful experiments should be undertaken in order to determine the best choice of (r, c). Elemental parallel eigensolver is based on a parallelized version of MRRR [28, 25], namely EleMRRR.

Scalability and efficiency of parallel libraries — Implementing a scalable eigensolver implies that all the stages, even the less expensive ones, should have the same degree of scalability. We will briefly illustrate how scalability depends on many factors and may vary as a function of the size of the problem as well as the number of cores utilized. In the following we will show some example of scalability for ScaLAPACK and Elemental. The numerical results and the plots are extracted from the paper [25] by generous concession of the authors.



(a) Execution time (b) Parallel efficiency

Fig. 1: *Strong scalability for the computation of all eigenpairs with matrices of size 20,000. Red and green lines corresponds to ScaLAPACK eigensolvers based on the tridiagonal D&C and MRRR. The blue line corresponds to the parallel MRRR-based eigensolver implemented in Elemental. (By courtesy of Petschow and Bientinesi)*

The strong scaling plots shows that there is not much difference between the three parallel eigensolvers up to 512 cores. At that point ScaLAPACK solvers performance degrades dramatically due to the MPI communication design decision: while the elemental distribution of Elemental does not force an algorithmic block, ScaLAPACK blocked communication hamper the scalability of the algorithm implementation over a certain number of cores. This is more evident by looking at the parallel efficiency which degrades quite dramatically for ScaLAPACK solvers above 512 cores, where the reference point is chosen to be 64 cores. Observe the perfect efficiency of `PZHEEVD` and `PZHEEVR` for 16 nodes (equivalent to 128 cores) due to the use of the very efficient ScaLAPACK `PZHENTRD` routine. The plots for weak scalability shows how Elemental scales much better for larger size eigenproblems than either ScaLAPACK D&C and MRRR. This result implies an almost perfect parallel efficiency and, consequently the possibility to solve increasingly bigger eigenproblems in a still reasonable amount of time.
The superior performance of the Elemental eigensolver (EleMRRR) is due to two main factors: 1) the elemental distribution strategy among processes favor a more fine coarse communication pattern which is close to optimal and 2) the tridiagonal eigensolver has been parallelized using a dynamic allocation of the load among all processes. The latter characteristic allows EleMRRR

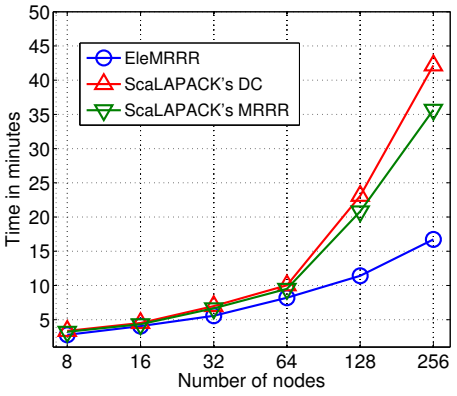


Fig. 2: *Weak scalability for the computation of all eigenpairs with matrices of increasing size. Red and green lines corresponds to ScaLAPACK eigensolvers based on the tridiagonal D&C and MRRR. The blue line corresponds to the parallel MRRR-based eigensolver implemented in Elemental. (By courtesy of Petschow and Bientinesi)*

to scale also when using thousands of cores. The same cannot be said for ScaLAPACK D&C tridiagonal stage: the fraction of time spent on this stage by the eigensolver goes from 4.5% on 64 cores to 41% on 2048 cores determining its drop in performance.

It is important to point out that both ELPA and EleMRRR are quite recent additions and are still not used by the majority of DFT codes dealing with dense eigenvalue problems. On the opposite the use of the more established ScaLAPCK is almost ubiquitous. This seem quite in contradiction with the desire to simulate physical systems with a higher number of atoms, which in turn needs a more efficient and scalable eigensolver. The main reason behind the lack of change is to be found in the need of a profound change for the data distribution pattern. From this point of view ELPA requires much less man-programming hours than Elemental. Despite this increased handicap, future DFT codes, which want to efficiently run on massive parallel architectures, will inevitably have to step up and face the initial investment.

4 Iterative eigensolvers

As mentioned at the beginning of section 3, for a sparse eigenproblem it is advisable to use an eigensolver which preserves the structure of the sparsely populated matrices which define it. It is important to notice that a matrix is defined as sparse when the number of non-zero entries (nnz) is less than a few percentage points of n^2. This is particularly important because these matrices are usually stored in memory in sparse format: only non-zero entries and their indices are specified. Consequently saving in memory only the nnz data frees up quite a bit of memory. Iterative methods do not attempt to directly diagonalize the matrix but instead strive at determining the eigenspace (or a subspace of the eigenspace) of the problem. This result is attained by repeadetly multiplying one (many) trial vector(s) with the matrix defining the HEVP[10] and in some way aligning it to the dominant eigenvector (eigenspace). This idea is at the base of the so-called power method and provides the basic principle on top of which all other iterative methods are built.

[10]The GHEVP is usually treated by reducing it to standard form even if there are several possible alternatives.

The power method relies on the basic concept that any random vector v can be decomposed as a linear combination $v = \sum_j \gamma_j x_j$ of the eigenvectors basis set $\{x_1, \ldots, x_n\}$ with $|\lambda_1| > |\lambda_2| > |\lambda_3| > \ldots$. The repeated multiplication of A on v from the left results in

$$Av = \sum_j \lambda_j \gamma_j x_j \quad \Longrightarrow \quad v^{(k)} = A^k v = \sum_j \lambda_j^k \gamma_j x_j = \lambda_1^k \left[x_1 + \sum_{j \geq 2} \left(\frac{\lambda_j}{\lambda_1} \right)^k \gamma_j x_j \right].$$

For large enough k the eigenvector with largest eigenvalue dominates over the others and $v^{(k)}$ converge to x_1 at the rate with which all the coefficients $\left| \frac{\lambda_j}{\lambda_1} \right|^k$ become negligible.
Since iterative methods are all roughly based on the power method, their effectiveness is grounded in multiple repetitions of matrix-vector multiplications. As such, iterative methods work on the vectors and not on the matrix, thus maintaining intact its sparse structure. While preserving the matrix sparse structure, efficient matrix-vector multiplications cannot rely on BLAS and need to be optimized for the specific problem or class of problems. This characteristic makes very difficult to know a priori the complexity of an iterative eigensolver. This is the more so since one does not know in advance when a trial vector would converge to an eigenvector.
In theory, there is nothing that prevents iterative methods to be used on dense matrices. In fact if the dense matrix is not too large and the desired fraction of the spectrum is very small ($< 1\%$), it is common belief that iterative methods could still be competitive, performance-wise, with direct ones. As long as the number s of matrix-vector multiplications required for convergence of the residuals is less than the inverse of the fraction of spectrum desired f, the iterative solver complexity $\mathcal{O}(s * f * n^2) \lesssim \mathcal{O}(n^3)$. In other words the iterative solver is no more complex than a direct one. We will see that this fact can be exploited in the case of some tailored eigensolvers, an example of which is described in Sec. 5.
Another issue to address with iterative solvers is the marked distinction between solving for the GHEVP and the HEVP. This difference has its source in the lack of a clear path which takes the GHEVP and transforms it to an HEVP. In general one would like to preserve the structure of the matrix B and so avoid expensive factorizations which would inevitably lead to a dense factor L and spoil the sparsity of the problem. Avoiding factorizations depends very much on how well-conditioned is the B matrix. If the condition number is low one could "invert" B and solve for $B^{-1}Ax = \lambda x$. On the contrary if $\kappa(B)$ is quite large, factorizations are unavoidable and the first two stages of the direct method are used to reduce the problem to standard form. In these cases it is customary to rely on incomplete factorization which limit the density of L and may still produce good enough solutions. The rest of this section will not enter in the merit of describing how to solve GHEVP and exclusively deal with solving HEVP with iterative eigensolvers.
The large class of iterative eigensolvers can be divided in 3 major groups. Subspace iteration based methods, Krylov-based methods, Davidson-based methods.

4.1 Three classes of iterative methods

Subspace iteration methods — Subspace iteration is just a generalization of the power method. Instead of iterating the multiplication of A with v ($v^{(i)} = Av^{(i-1)}$) one can iterate on a space of dimension larger than one $V \in \mathbb{C}^{n \times k}$. Iterating on such initial space would, in general, lead to a *dominant* k-dimensional subspace $\mathcal{U}$ associated with the first k eigenvalues $|\lambda_1| \geq \cdots \geq |\lambda_k|$. The global convergence ratio now clearly depends on the magnitude of the largest among the coefficients required to be negligible $\left| \frac{\lambda_{k+1}}{\lambda_k} \right|$.

In practice, if one use subspace iteration in its purest form $V^{(i)} = AV^{(i-1)}$, each single $v_j^{(i)} \in V^{(i)}$ becomes mostly aligned with the eigenvector corresponding to $\lambda_{\max}$. After a sufficient number of iterations the subspace could become rank-deficient leading to multiple copies of the same eigenvectors. In order to avoid this undesirable effect, subspace iteration is usually used in combination with a shift $A - \sigma I$. The dominant eigenvector of the shifted matrix would be the one corresponding to the eigenvalue closer to σ. By using a different shift at each iteration distinct eigenspaces can be enhanced at each iteration

$$V^{(i)} = (A - \sigma_i I) \cdots (A - \sigma_1 I)V$$

To work properly this method assumes the a priori knowledge of shifts $\{\sigma_1, \ldots, \sigma_k\}$ close enough to the true eigenvalues. Such knowledge is not often available but can be inferred by evaluating the Ritz values after a certain number of non-shifted iterations.
The procedure described above is equivalent to using for a single iteration a polynomial $p(t) = (t - \sigma_i) \cdots (t - \sigma_1)$ having zeros close to the k dominant eigenvalues. In addition to generalize the subspace iteration to polynomials one can orthonormalize the resulting vectors using the Gram-Schmidt procedure

$$p(A)V^{(i-1)} = [p(A)v_1^{(i-1)}, \ldots, p(A)v_k^{(i-1)}] \longrightarrow [q_1^{(i)}, \ldots, q_k^{(i)}] = Q^{(i)} \text{ with } (Q^{(i)})^\dagger Q^{(i)} = I.$$

The convergence to the solutions of the iterates can be tricky. In general computing the error (as the distance between successive vectors) may not guarantee that when the error is small we reached convergence. A better criterion is to check the residuals for the approximated eigenvectors and declare the eigenpairs converged when residuals are below the required tolerance. In general one measure the rate of convergence of the process by looking at the difference between the residuals across two iterations. If this difference decrease linearly one refers to it as linear convergence. It is a known results of numerical linear algebra that subspace iteration can at best converge linearly.

Krylov subspace methods — Krylov-based methods refer to a specifically constructed sequence of subspaces approximating an invariant subspace of the entire spectrum. The relative space is called Krylov and is indicated by $\mathcal{K}_k(A, v)$. The ingenuity of the Krylov method is in building a sequence of subspaces of increasing dimensionality (as opposed to the fixed dimensionality of subspace iteration). In practice the subspace is characterized directly by an orthonormal space which is the by-product of the construction.
One start with a normalized vector $u = u_1 \equiv \mathcal{K}_1(A, u)$ and compute u_2 as

$$u_2 h_{2,1} = Au_1 - h_{1,1}u_1 \quad ; \quad \mathcal{K}_2(A, u) = \text{span}\{u_1, u_2\}$$

where $h_{1,1}$ is chosen so as to make u_1 orthogonal to u_2, and $h_{2,1}$ is just a scale factor. The whole process is then repeated j times to find $\mathcal{K}_j(A, u) = \text{span}\{u_1, \ldots, u_j\}$. At the $j + 1$ step, one first checks if $Au_j \in \text{span}\{u_1, \ldots, u_j\}$. If this is the case, the coefficent $h_{i,j}$ should be chosen such that $Au_j = \sum_{i=1}^{j} u_i h_{i,j}$. In this way $h_{j+1,j} = 0$ and u_{j+1} is indeterminate. At this point the Krylov process terminates and the resulting subspace is invariant.
In general the k-step of the Krylov process can be represented in matricial form as

$$AU_k = U_k H_{k+1,k}$$

where the matrix U_k collects all the u vectors, and $H_{k+1,k}$ is a matrix accumulating all $h_{i,j}$. If one defines the upper Hessenberg matrix deleting the bottom row of $H_{k+1,k}$ then the process is better described by

$$AU_k = U_k H_k + u_{k+1} h_{k+1,k} e_k^T.$$

Whenever $h_{k+1,k} = 0$ then $A = UHU^{-1}$ which clearly shows that the matrix H corresponds to the eigenproblem A reduced to the invariant subspace $\mathcal{K}_k(A, u)$. Even when the Krylov process is only partially completed it produces a quasi-similarity transformation.
Thus even if $h_{k+1,k}$ is not equal to zero, the Krylov process leads to a subspace which contains good approximants to the eigenvectors of some of the peripheral eigenvalues of A. This property constitutes the basis of the the so-called Arnoldi method of *implicit restarts*. It has been successfully implemented by Sorensen et Al. in the package ARPACK [11]. The aim of the Arnoldi procedure is to keep $m \ll k$ vectors of the Krylov subspace which are rich in the components of the good approximants to the eigenvectors, discard the rest and start the Krylov process again from these m vectors. This is achieved through a GR process whose details are not important for the purpose of this chapter. We remit the reader to the vast literature on the subject [29].

Algorithm 1 Symmetric Lanczos algorithm with re-orthogonalization

Require: Matrix A of the HEVP and initial vector u
Ensure: NEV wanted eigenpairs (λ, x).

$u_1 \leftarrow u/\|u\|$
$U_1 \leftarrow [u_1]$
for $j = 1, 2, 3, \ldots \rightarrow$ NEV (λ, x) **do**
  $u_{j+1} \leftarrow A u_j$
  $\alpha_j \leftarrow u_j^\dagger u_{j+1}$
  $u_{j+1} \leftarrow u_{j+1} - \alpha_j u_j$
  if $j > 1$ **then**
    $u_{j+1} \leftarrow u_{j+1} - \beta_{j-1} u_{j-1}$
  end if
  $\delta_{1:j} \leftarrow U_j^\dagger u_{j+1}$
  $u_{j+1} \leftarrow u_{j+1} - U_j \delta_{1:j}$ ▷ RE-ORTHOGONALIZATION
  $\alpha_j \leftarrow \alpha_j + \delta_j$
  $\beta_j \leftarrow \|u_{j+1}\|$
  if $\beta_j = 0$ **then**
    Flag span$\{u_1, \ldots, u_j\}$ is invariant
    Exit
  end if
  $u_{j+1} \leftarrow u_{j+1}/\beta_j$
  $U_{j+1} \leftarrow [U_j \; u_{j+1}]$
end for

Since the Krylov process preserves the symmetry of the matrix A, in the case of the Hermitian (or symmetric) eigenvalue problem the matrix H_k is actually tridiagonal. The net result is that the whole process of building H is actually reduced to a 3-term recurrence relation

$$u_{j+1}\beta_{j+1} = Au_j - \alpha u_j - \beta_{j-1} u_{j-1}$$

In this case the Krylov-Arnoldi process is called Lanczos process (see Alg. 1)

The Krylov process is an instance of the the Gram-Schmidt method and as such is vulnerable to roundoff errors. Over the course of several steps executed in floating point arithmetic, the orthogonality of the vectors steadily deteriorate. The standard remedy, as suggested by Kahan in an unpublished work, is to orthogonalize twice. This practice has also been confirmed by numerical analysis [30]. Thus the safest way to preserve orthogonality is to save all the vectors and orthogonalize against them.
By re-orthogonalizing, the Lanczos process looks very similar to the Arnoldi, so one may wonder why bothering with the 3-term recurrence relation. The simple answer is that the Lanczos process has the advantage of preserving the structure of the matrix H leaving it symmetric like the original A. Then the restart can be performed through a symmetric QR algorithm which also keep the structure intact and has a lower complexity than the full GR.

Davidson methods — The original Davidson method was devised by the homonymous author to compute the lowest energy levels and corresponding wave functions for eigenvalues problems arising in quantum chemistry. Davidson algorithm builds a subspace of increasing dimensionality by adding a new vector at each step in a way similar to the Lanczos algorithm. The main difference lies in the choice of vector: instead of being extracted by Au_{k-1} the additional vector is obtained with a correction equation for the residual $r_{k-1} = (Au_{k-1} - \hat{\lambda}_{k-1}u_{k-1})$ sometimes also referred as diagonal *preconditioning step*.
Several are the variants of the Davidson method, some of which are also implemented in a block version. All these can be classified in two major groups: Generalized Davidson (GD) and Jacobi-Davidson (JD) methods. In its original and simplest form the algorithm would look only for the largest or smallest eigenpair. In Alg. 2 we present a rather simple formulation which shows the major differences with Lanczos are in lines 5 and 12. The simplest Davidson method

Algorithm 2 Davidson algorithm

Require: Matrix A of the HEVP and initial vector u
Ensure: largest eigenpair $(\lambda_{\max}, x)$.

1: $u_1 \leftarrow u/\|u\|$
2: $U_1 \leftarrow [u_1]$
3: **for** $j = 1, 2, 3, \ldots \rightarrow$ NEV (λ, x) **do**
4: $H_j \leftarrow U_j^\dagger A U_j$ ▷ RAYLEIGH-RITZ QUOTIENT
5: Compute largest eigenpair $(\hat{\lambda}_j, \hat{y}_j)$ of H_j
6: Compute Ritz vector $x_j \leftarrow U_j y_j$
7: Compute the residual $r_j \leftarrow (\hat{\lambda}_j I - A)x_j$
8: **if** $r_j <$ TOL **then**
9: $(\lambda_{\max}, x) \leftarrow (\hat{\lambda}_j, \hat{y}_j)$
10: Exit
11: **end if**
12: Correction equation $t_{j+1} \leftarrow (\hat{\lambda}_j I - D)^{-1} r_j$
13: $\delta_{1:j} \leftarrow U_j^\dagger t_{j+1}$
14: $t_{j+1} \leftarrow t_{j+1} - U_j \delta_{1:j}$ ▷ ORTHOGONALIZATION
15: $t_{j+1} \leftarrow t_{j+1}/\|t_{j+1}\|$
16: $U_{j+1} \leftarrow [U_j\ t_{j+1}]$
17: **end for**

does not rely on any kind of restart but keeps building a subspace and then constructing the

Rayleigh-Ritz quotient out of which it extracts an approximant to the largest eigenpair. The subspace is incremented by solving a "preconditioned" linear system associated with the HEVP where the known term is the residual of the approximant and D is the main diagonal of A.
Davidson method can be straightforwardly generalized to a block implementation where the U_j is incremented with a block of vectors and a block of eigenpairs is sought after. The block adaptation is particularly relevant when the resulting code are ported over parallel computing architectures. What is not obvious is how to generalize the correction equation on line 12. In general this equation can be written as $C_{j+1}t_{j+1} = -r_j$ and solved with a low level of accuracy. The operator C_{j+1} can be as simple as $(A-\hat{\lambda}_j I)$ or as complicated as $(I-x_j x_j^\dagger)(A-\hat{\lambda}_j I)(I-x_j x_j^\dagger)$. The latter choice is at the base of the JD method: by solving the correction equation orthogonally to the Ritz vector x_j, JD avoids the well-known effect of *stagnation*. In addition to an improved correction equation a preconditioner can be used. In the case of JD the inverse of such preconditioner needs to be inverted orthogonally to x_j. Additionally the Davidson methods could be restarted for efficient use of the memory and better convergence of the subspace approximants.
It is worth to mention an algorithm which does not quite fit the above grouping: the Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG). Similar to a block version of GD the LOBPCG can deal directly with GHEVP. Developed in 2001 by Knyazev [31, 32], this algorithm uses a locally optimized version of a three-term recurrence relation for the preconditioned conjugate gradient method. In practice the Rayleigh-Ritz method is used for the eigenpencil on a trial subspace generated by the current guess for the Ritz vector, the preconditioned residual, and a third Ritz vector built by maximizing the Rayleigh quotient. Knyazev implemented a block version of the algorithm where the three-term relation is generalized for a block of vectors. LOBPCG can deal directly with GHEVP only if B is well-conditioned. When $\kappa(B)$ has a large value the performance of the algorithm deteriorates quite rapidly and reduction to standard form is more stable [33].

4.2 Libraries

ARPACK and PARPACK — The first and foremost best known library of iterative eigensolver is the Arnoldi Restarted Package (ARPACK). This library is based upon an algorithmic variant of the Implicit Restarted Arnoldi Method (IRAM) which reduces to the Implicit Restarted Lanczos Method (IRLM) for Hermitian or symmetric eigenproblems. Both variants are just instances of the general Implicitly Shifted QR technique applied to the the Krylov-Arnoldi process. ARPACK is conceived to solve for large scale symmetric, or non-symmetric eigenvalue problems. The software is designed for sparse and structured matrices such that the matrix-vector product Au would involve only $\mathcal{O}(n)$ operations instead of the standard $\mathcal{O}(n^2)$.
The collection of subroutines making up the library are written in Fortran77. For many standard problems, there is no need for an explicit matrix factorization. Only the action of the matrix A on a vector is required. The software is aimed at computing a small fraction of the total number of eigenvalues with either largest real part or largest magnitude. Storage requirements are on the order of $n \times s$ with s being the number of required eigenpairs. No extra auxiliary storage is required. A set of Schur basis vectors for the desired s-dimensional eigenspace is computed which is numerically orthogonal to working precision. Numerically accurate eigenvectors can be computed on request.
The PARPACK software package realizes the parallelization of the ARPACK library for distributed memory architectures. It has been designed so as to provide the user with a Single Pro-

gram Multiple Data (SPMD) template. The reverse communication interface, which is the most important design feature of ARPACK, has motivated the parallelization strategy. The interface enables PARPACK to be internally parallelized by avoiding to impose predetermined parallel decompositions on A and on the user-provided matrix-vector product. The call to PARPACK preserve the same structure as ARPACK, the only difference consisting in the inclusion of the Basic Linear Algebra Communication Subprograms context. The net result is outlined in 3 main steps: 1) replicating H_j on every processor, 2) distributing (blocked by rows) U_j on a 1-dimensional processor grid, 3) distributing the workspace accordingly. The greater part of the communication takes place during the Gram-Schmidt orthogonalization and possibly in the user supplied matrix-vector multiplication. Clearly the intrinsic serial nature of the algorithm does not lend itself to parallelization on massive number of processors when the parallel efficiency degrades substantially.

SLEPc — The Scalable Library for Eigenvalue Problem Computations (SLEPc) [34], is a software library for the solution of large sparse eigenproblems on parallel computing architectures. The majority of SLEPc routines are intended to be used for the solution of both generalized and standard eigenvalue problems in their linear or non-linear version. SLEPc provides a large set of different methods and focus on the choice of the most appropriate one in relation to the application which generated it. Most of the methods offered by the library are indeed subspace methods. The default eigensolver is a slight modification of the classic Krylov process named Krylov-Schur method [35, 36]. The library also offers routines implementing JD and Conjugate Gradient methods as well as an interface to several external packages like ARPACK, BLZPACK, TRLAN, BLOPEX, and PRIMME. The library does not limit itself to the solution of Hermitian problems but it extends also to non-symmetric and more general complex-valued eigenproblems. Among its other functionalities, it provides routines for Singular Value Decompositions and tools of spectral transformations such as shift and invert, etc..
SLEPc is built on top of the Portable Extensible Toolkit for Scientific Computation (PETSc) [37]. As an extension to PETSc toolkit, SLEPc inherits a variety of tuned data structures, multivector operations, matrix-vector and preconditioning operators, but it cannot run as stand-alone with applications that do not use PETSc. This means that PETSc must be previously installed in order to use SLEPc. This library enforces the same programming paradigm as PETSc making it quite easy to manipulate for users already acquainted with the use of the latter. Users who are not familiar with PETSc are invited to first get accustomed to the basic concepts of such framework before endeavor in the use of SLEPc.

PRIMME and BLOPEX — The PReconditioned Iterative Multi Method Eigensolver (PRIMME) [38] is a software package for the solution of large, sparse Hermitian and real symmetric standard eigenvalue problems. PRIMME constitutes a significant effort towards the realization of a robust and effective code for the solution of large and difficult eigenproblems when matrix factorization is not a viable option and the user has to rely only on matrix-vector operations and, possibly, a preconditioning operator. PRIMME developers stress that the library design strategy is to provide a flexible eigensolver with the following salient characteristics: 1) preconditioned eigen-methods converging near optimally under limited memory, 2) the maximum robustness possible without matrix factorization, 3) flexibility in mixing and matching among most currently known features, 4) efficiency for most architectural layout, and 5) a friendly user interface that requires no parameter setting from end-users but permits full ex-

perimentation by experts. The algorithmic framework of PRIMME is built on top of the two near optimal methods derived from the class of Davidson algorithms: GD+k and JDQMR. It is remarkable that these two algorithms also provide the structure to show how other algorithms can be parameterized within this framework.
The Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) [32] is a package, written in C and MATLAB/OCTAVE, that includes an eigensolver implemented with the Locally Optimal Block Preconditioned Conjugate Gradient Method. Its prominent characteristics are: 1) a matrix-free iterative method for computing several extreme eigenpairs of symmetric positive generalized eigenproblems, 2) a user-defined symmetric positive preconditioner, 3) robustness with respect to random initial approximations, variable preconditioners, and ill-conditioning of the overlap matrix. BLOPEX supports parallel MPI-based computations. BLOPEX is incorporated in the HYPRE software and is available as an external package to the PETSc framework.

Anasazi — Anasazi [39] is a well engineered package, with several features that boost robustness and efficiency. Anasazi provides a generic interface to a collection of algorithms for solving large-scale eigenvalue problems. The package implements three methods: a version of the block Krylov-Schur algorithm, a variant of the LOBPCG method with orthogonalization to avoid stability issues, and a block GD method for solving standard and generalized real symmetric and Hermitian eigenvalue problems. All methods are implemented in block variants in order to increase robustness for obtaining multiple eigenvalues and to take advantage of the increased data locality in block matrix-vector, pre-conditioning, and BLAS operations. Even if the total number of matrix-vector multiplications increases in block implementations, for appropriate block sizes this effect is usually balanced by better cache performance.
Anasazi is an interoperable software since both the matrix and vectors are defined as opaque objects so that only knowledge of the matrix and vectors via elementary operations is necessary. As such Anasazi implementations are accomplished via the use of interfaces. Current available interfaces include Epetra. Consequently any libraries that understand Epetra matrices and vectors may be used in conjunction with Anasazi. One of the goals of Anasazi is to allow the user the flexibility in specifying the data representation for the matrix and vectors and so leverage pre-existing software investment. Anasazi is part of the Trilinos framework that includes highly optimized linear algebra operations.

DFT codes implementations — Many of the algorithms previously illustrated have been implemented in variants specifically tailored to DFT codes. Below we give some interesting examples which are far from being exhaustive but can give the reader a flavor of the variety with which iterative methods can be used.
The ABINIT code [40] uses a variant of subspace iteration together with an iterative scheme which improves the initial choice of vectors for the subspace through a form of conjugate gradient algorithm. Orthogonality of the resulting vectors is ensured by a Rayleigh-Ritz method applied to all the subspace each time one eigenvector approximant is modified. ABINIT includes also a block version of the algorithm where parallelism is offered across the several vectors of the block as well as a variant minimizing the residual norm of the subspace vectors.
Among the several diagonalization methods it offers, the Vienna Ab-initio Simulation Package (VASP) [12], uses a Davidson approach with a form of block preconditioning. This method is recommended as a robust alternative to the other methods, though it is also mentioned as being

more costly in some cases. The PARSEC code [41] also uses a modified version of the block Davidson code where the correction equation is preconditioned with a Chebyshev polynomial filter so as to enhance components of the new vectors which will be used to increase the subspace. This method also offers a double restart procedure with an outer and an inner restart loop. The outer restart loop is very similar to a standard implicit restart in keeping the best approximants among the computed subspace. The inner restart loop instead allows the addition of a new block of approximate vectors as soon as some of the sought after eigenpairs have converged. This version of the Davidson method succeeds in better deflating converged vectors and has the added ability to accept approximate solutions in place of the standard augmentation vectors.

Many DFT codes offer some sort of generalization or modified version of a direct inversion of the subspace iteration (DIIS), a method firstly proposed by Pulay [42, 43] in the early '80s. DIIS is a form of Krylov subspace method where an initial subspace is improved through an iterative scheme which individually minimize the residuals of the eigenvectors approximants. In this approach, there is no need to orthogonalize each vector against all others after each update to the basis vectors.

Overall it has been observed by many that Davidson-based algorithms are more robust than methods based on local optimization (like DIIS or Conjugate Gradient). This observation is not a unanimous viewpoint. For example, developers of PWscf and VASP seem to recommend direct minimization, in spite of its less favorable speed. In the end implementations of each specific algorithm is a key factor. With proper implementation, a Davidson- or Krylov-based approach should be vastly superior to direct minimization.

5 DFT-tailored algorithms: an example

As mentioned in the Introduction section, DFT-based methods lead to the self-consistent solution of linearized eigenvalue problems. In other words the non-linear eigenvalue problem generated by the KS equation is solved by a sequence of eigenproblems whose solution is increasingly closer to the one of the original non-linear problem. In practice one starts with a GHEVP $P^{(1)} : A^{(1)}x = \lambda B^{(1)}x$, solves for it, use the solutions to generate the new $P^{(2)}$ so on and so forth. In the end the computational scientist needs to solve a sequence of eigenproblems $\{P^{(1)}, \ldots, P^{(\ell)}, \ldots, P^{(N)}\}$.

It is reasonable to assume — and it has been shown numerically in [44] — that the problems in the sequence are correlated to each other. Thus a particularly tailored eigensolver could take advantage of the correlation to improve performance and scalability. In this section we show an example of such approach specifically designed for DFT methods based on the LAPW basis set. The algorithm choice for this specific set-up is rather peculiar: an iterative eigensolver is selected to solve dense eigenproblems, a pairing which is, in principle, unfavorable. We report below its properties and the numerical performance it achieves.

Chebyshev Filtered Subspace Iteration — Subspace Iteration complemented with a Chebyshev polynomial filter is a well known algorithm in the literature [45]. A version of it was recently developed for a real space dicretization of DFT by Chelikowsky et Al. [46, 47] and included in the PARSEC code. By using a polynomial filter on the initial block of inputed vectors the method experiences a high rate of acceleration. Since the block of vectors spanning

the invariant subspace could easily become linearly dependent the subspace iteration is usually complemented with some re-orthogonalization procedure.
The Chebyshev Filtered Subspace Iteration (ChFSI) algorithm described here is a slightly more sophisticated version of the basic accelerated subspace iteration. This variant is specifically tailored for DFT-like sequences of eigenproblems and it has been developed with the LAPW discretization in mind. In particular the ChFSI algorithm takes advantage of the eigenvectors of the eigenproblem of the (ℓ)-SCF cycle and uses them as input to solve for the eigenproblem at the $(\ell+1)$-SCF cycle.
The whole algorithm is illustrated in the Algorithm 3 scheme. Notice that the initial input is not the initial GHEVP $A^{(\ell)}x = \lambda B^{(\ell)}x$ but its reduction to standard form $H^{(\ell)} = L^{-1}A^{(\ell)}L^{-\mathrm{T}}$ where $B^{(\ell)} = LL^{\mathrm{T}}$, and $\hat{Y}^{(\ell-1)}$ are the eigenvectors of $H^{(\ell-1)}$. ChFSI uses few Lanczos iterations (`line 1`) so as to estimate the upper limit of the eigenproblem spectrum [48]. This estimate is necessary for the correct usage of the filter based on Chebyshev polynomials [45]. After the Chebyshev filter step (`line 3`) the resulting block of vectors is re-orthonormalized using a simple QR factorization (`line 4`) followed by a Rayleigh-Ritz procedure (`line 5`). At the end of the Rayleigh-Ritz step eigenvector residuals are computed, converged eigenpairs are deflated and locked (`line 13`) while the non-converged vectors are sent again to the filter to repeat the whole procedure.

Algorithm 3 Chebyshev Filtered Subspace Iteration with locking

Require: Matrix $H^{(\ell)}$ of the DGEVP reduced to standard form, approximate eigenvectors $\hat{Y}^{(\ell-1)} \doteq \left[\hat{y}_1^{(\ell-1)}, \ldots, \hat{y}_{\text{NEV}}^{(\ell-1)}\right]$ and eigenvalues $\lambda_1^{(\ell-1)}$ and $\lambda_{\text{NEV}+1}^{(\ell-1)}$.
Ensure: Wanted eigenpairs (Λ, Y) .

1: Estimate the largest eigenvalue. ▷ LANCZOS
2: **repeat**
3: Filter the vectors, $\hat{Y} = C_m(\hat{Y})$. ▷ CHEBYSHEV FILTER
4: Re-orthonormalize $\hat{Y}$. ▷ QR ALGORITHM
5: Compute Rayleigh quotient $G = \hat{Y}^{\dagger}H^{(\ell)}\hat{Y}$. ▷ RAYLEIGH-RITZ (Start)
6: Solve the reduced problem $G\hat{W} = \hat{W}\hat{\Lambda}$.
7: Compute $\hat{Y} = \hat{Y}\hat{W}$. ▷ RAYLEIGH-RITZ (End)
8: **for** $i =$ converged $\rightarrow$ NEV **do** ▷ DEFLATION & LOCKING (Start)
9: **if** $r(\hat{Y}_{:,i}, \hat{\Lambda}_i) <$ TOL **then**
10: $\Lambda = \left[\Lambda\ \hat{\Lambda}_i\right]$
11: $Y = \left[Y\ \hat{Y}_{:,i}\right]$
12: **end if**
13: **end for** ▷ DEFLATION & LOCKING (End)
14: **until** converged $\geq$ NEV

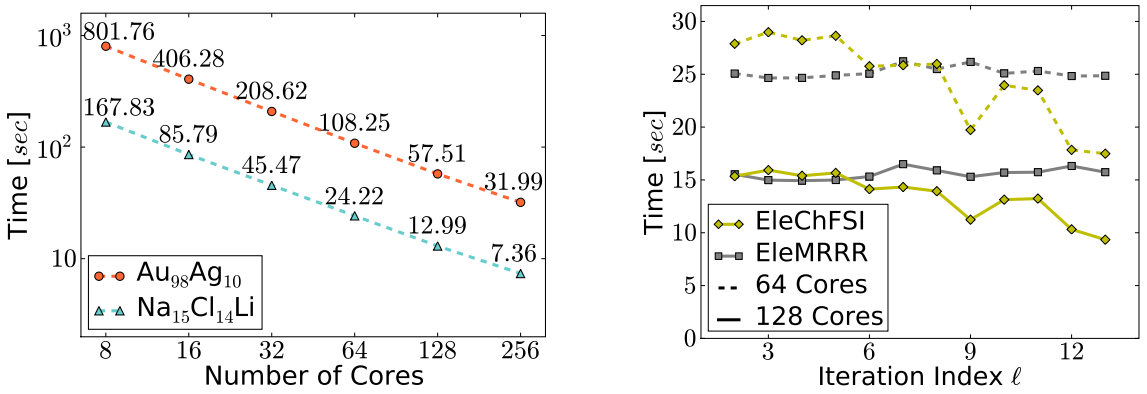
The Chebyshev polynomial filter is at the core of the algorithm. The vectors $\hat{Y}$ are filtered exploiting the 3-terms recurrence relation which defines Chebyshev polynomials of the first kind

$$C_{m+1}(\hat{Y}) = 2\ H\ C_m(\hat{Y}) - C_{m-1}(\hat{Y}) \quad ; \quad C_m(\hat{Y}) \doteq C_m(H)\cdot\hat{Y}. \tag{4}$$

This construction implies all operations internal to the filter are executed through the use of ZGEMM, the most performant among BLAS 3 routines. Since roughly 90% of the total CPU

time is spent in the filter, the massive use of ZGEMM makes ChFSI quite an efficient algorithm and potentially a very scalable one. The parallel MPI version of ChFSI (EleChFSI) is implemented within the Elemental library. The reduced eigenproblem in the Rayleigh-Ritz step is solved using a parallel implementation of the MRRR eigensolver (EleMRRR).

Performance and scalability — In the plots below we report on the scalability of EleChFSI and its performance when compared with the fastest direct method available on the market as shown in [49].
Plot (a) of Fig. 3 illustrate the strong scalability of EleChFSI showing a steady decrease of CPU time as the number of cores increases. The rate of reduction is practically the same for both atomic systems despite their size differ by more than 30%. This plot shows that EleChFSI is extremely efficient even when the ratio of data per processor is not optimal. This result is due both on the re-use of eigenvectors of the previous SCF and on the extensive use of BLAS within the Elemental framework.



(a) EleChFSI Strong scalability.

(b) EleChFSI performance.

Fig. 3: *EleChFSI strong scalability and performance. In plot (a) the size of the eigenproblems are kept fixed while the number of cores is progressively increased. Eigenproblems of size* $n = 13,379$ *and* $n = 9,273$ *are shown. In plot (b) EleChFSI is compared with EleMRRR on eigenproblems of increasing self-consistent cycle index* ℓ *for a sequence of eigenproblems with* $n = 9,273$.

Compared to direct solvers, EleChFSI promises to be quite competitive. Depending on the number of eigenpairs computed, the algorithm implementation is even faster than EleMRRR. In plot (b) of Fig. 3 EleChFSI is already faster than EleMRRR for half of the eigenproblems in the sequence (64 cores). When the tests are repeated with 128 cores EleChFSI is inequivocably the faster of the two algorithms. Since the fraction of the spectrum computed in plot (b) is $\sim 3\%$, Fig. 3 shows that EleChFSI scales better than EleMRRR and is more performant when the sought number of eigenpairs is not too high.

References

[1] P. Hohenberg, Physical Review **136**(3B), B864 (1964).

[2] W. Kohn and L. J. Sham, Phys.Rev. **140**, A1133 (1965).

[3] S. Huzinaga, The Journal of Chemical Physics **42**(4), 1293 (1965).

[4] J. A. Pople and W. J. Hehre, Journal of Computational Physics **27**(2), 161 (1978).

[5] E. Wimmer, M. Weinert, and A. J. Freeman, Physical Review B **24**(2), 864 (1981).

[6] H. J. F. Jansen and A. J. Freeman, Physical Review B **30**(2), 561 (1984).

[7] J. Chelikowsky, N. Troullier, K. Wu, and Y. Saad, Physical Review B **50**(16), 11355 (1994).

[8] S. White, J. Wilkins, and M. Teter, Physical Review B **39**(9), 5819 (1989).

[9] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. A. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, and A. Bergman, The Journal of Chemical Physics **129**, 014109 (2008).

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, *et al.*, *ScaLAPACK Users' Guide* (Society for Industrial and Applied Mathematics, 1987).

[11] R. Lehoucq, D. Sorensen, and C. C. Yang, *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems* (Society for Industrial and Applied Mathematics, 1998).

[12] G. Kresse and J. Furthmüller, Physical Review B **54**(16), 11169 (1996).

[13] Y. Saad, Y. Zhou, C. Bekas, M. L. Tiago, and J. Chelikowsky, physica status solidi (b) **243**(9), 2188 (2006).

[14] G. M. Amdahl, in *the April 18-20, 1967, spring joint computer conference* (ACM Press, New York, New York, USA, 1967), p. 483.

[15] J. L. Gustafson, Communications of the ACM **31**(5), 532 (1988).

[16] G. H. Golub and C. F. Van Loan, *Matrix Computations* (Johns Hopkins Univ., 2012).

[17] C. H. Bischof, B. Lang, and X. Sun, ACM Transactions on Mathematical Software **26**(4), 581 (2000).

[18] D. S. Dodson and J. G. Lewis, ACM Signum Newsletter **20**(1), 19 (1985).

[19] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Transactions on Mathematical Software **14**(1), 18 (1988).

[20] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, ACM Transactions on Mathematical Software **16**(1), 1 (1990).

[21] K. Goto and R. A. v. d. Geijn, ACM Transactions on Mathematical Software **34**(3), 1 (2008).

[22] K. Goto and R. van de Geijn, ACM Transactions on Mathematical Software **35**(1), 1 (2008).

[23] R. C. Whaley and J. J. Dongarra, *Automatically tuned linear algebra software* (IEEE Computer Society, 1998).

[24] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel, SIAM Journal on Scientific Computing **30**(3), 1508 (2008).

[25] M. Petschow, E. Peise, and P. Bientinesi, SIAM Journal on Scientific Computing **35**(1), C1 (2013).

[26] T. Auckenthaler, H. J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems, Journal of Computational Science **2**(3), 272 (2011).

[27] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, ACM Transactions on Mathematical Software (TOMS) **39**(2) (2013).

[28] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn, SIAM Journal on Scientific Computing **27**(1), 43 (2005).

[29] D. S. Watkins, *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods* (Society for Industrial and Applied Mathematics, 2007).

[30] L. Giraud, J. Langou, M. Rozložník, and J. v. d. Eshof, Numerische Mathematik **101**(1), 87 (2005).

[31] A. V. Knyazev, SIAM Journal on Scientific Computing **23**(2), 517 (2001).

[32] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov, SIAM Journal on Scientific Computing **29**(5), 2224 (2007).

[33] E. Di Napoli and M. Berljafa, Computer Physics Communications **184**(11), 2478 (2013).

[34] V. Hernandez, J. E. Roman, and V. Vidal, ACM Transactions on Mathematical Software **31**(3), 351 (2005).

[35] G. W. Stewart, SIAM Journal on Matrix Analysis and Applications **23**(3), 601 (2002).

[36] G. W. Stewart, SIAM Journal on Matrix Analysis and Applications **24**(2), 599 (2002).

[37] S. Balay, W. Gropp, L. Curfman McInnes, and B. Smith, *PETSc 2.0 users manual: Revision 2.0.16* (Argonne National Laboratory, 1997).

[38] A. Stathopoulos and J. R. McCombs, ACM Transactions on Mathematical Software (TOMS) **37**(2), 21 (2010).

[39] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, ACM Transactions on Mathematical Software (TOMS) **36**(3), 13 (2009).

[40] X. Gonze, B. Amadon, P. M. Anglade, J. M. Beuken, F. Bottin, P. Boulanger, F. Bruneval, D. Caliste, R. Caracas, M. Côté, T. Deutsch, L. Genovese, *et al.*, Computer Physics Communications **180**(12), 2582 (2009).

[41] L. Kronik, A. Makmal, M. L. Tiago, M. M. G. Alemany, M. Jain, X. Huang, Y. Saad, and J. R. Chelikowsky, physica status solidi (b) **243**(5), 1063 (2006).

[42] P. Pulay, Chemical Physics Letters **73**(2), 393 (1980).

[43] P. Pulay, Journal of Computational Chemistry **3**(4), 556 (1982).

[44] E. Di Napoli, S. Blügel, and P. Bientinesi, Computer Physics Communications **183**(8), 1674 (2012).

[45] Y. Saad, *Numerical methods for large eigenvalue problems* (Siam, 2011).

[46] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, Journal of Computational Physics **219**(1), 172 (2006).

[47] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, Physical Review E **74**(6), 066704 (2006).

[48] Y. Zhou and R.-C. Li, Linear Algebra and its Applications **435**(3), 480 (2011).

[49] M. Berljafa and E. Di Napoli, arXiv.org (2013), `1305.5120v1`.