Fast solution of elliptic partial differential equations using linear combinations of plane waves

José M. Pérez-Jordá*
Departament de Química Física, Universitat d'Alacant, E-03080, Alacant, Spain
(Received 19 October 2015; published 17 February 2016)

Given an arbitrary elliptic partial differential equation (PDE), a procedure for obtaining its solution is proposed based on the method of Ritz: the solution is written as a linear combination of plane waves and the coefficients are obtained by variational minimization. The PDE to be solved is cast as a system of linear equations $\mathbf{Ax} = \mathbf{b}$, where the matrix $\mathbf{A}$ is *not* sparse, which prevents the straightforward application of standard iterative methods in order to solve it. This sparseness problem can be circumvented by means of a recursive bisection approach based on the fast Fourier transform, which makes it possible to implement fast versions of some stationary iterative methods (such as Gauss-Seidel) consuming $O(N \log N)$ memory and executing an iteration in $O(N \log^2 N)$ time, N being the number of plane waves used. In a similar way, fast versions of Krylov subspace methods and multigrid methods can also be implemented. These procedures are tested on Poisson's equation expressed in *adaptive coordinates*. It is found that the best results are obtained with the GMRES method using a multigrid preconditioner with Gauss-Seidel relaxation steps.

DOI: 10.1103/PhysRevE.93.023304

I. INTRODUCTION

The evaluation of the *Coulomb potential* (v) generated from a given electronic density (ρ) is a necessary step in any quantum mechanical calculation on atoms, molecules, or solids. This evaluation can be accomplished via Poisson's equation,

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} = -4\pi\rho. \tag{1}$$

For periodic systems [1], v is usually expressed as a linear combination of plane waves,

$$v(\mathbf{r}) = \sum_{\mathbf{k}} c_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r}}, \tag{2}$$

where $\mathbf{k}$ is a wave vector and $\mathbf{r} = (x, y, z)$. Plane waves are delocalized over the whole system, which make them unsuitable for describing the sharp oscillations experienced by Coulomb potentials in the neighborhood of chemical bonds and nuclei. One solution to this problem is the use of *pseudopotentials* [1]. Another approach is to express the plane waves not as functions of Cartesians coordinates but as functions of a new coordinate system $\mathbf{u} = \mathbf{u}(\mathbf{r})$,

$$e^{i\mathbf{k}\cdot\mathbf{u}}. \tag{3}$$

The new coordinates are called *adaptive coordinates* [2,3] because the map $\mathbf{u} = \mathbf{u}(\mathbf{r})$ can be adapted to the system under study. Adaptive coordinates have been employed in a number of solid-state or quantum chemical applications [2–21]. For example, they have been used [12] for pseudopotential-free all-electron calculations on diamond.

Despite the obvious appeal of these adaptive coordinates, they have a downside that complicates their use: the mathematical description of the problem becomes more complex with the new coordinates. For example, the neat expression of Poisson's equation in Cartesian coordinates given in Eq. (1) is transformed into a full-blown *elliptic* partial differential equation (PDE) when adaptive coordinates are used. Let us remind the reader that an elliptic linear second-order PDE takes the following general form [22]:

$$\sum_{i,j=1}^{n} a_{ij}(\mathbf{r}) \frac{\partial^2 f}{\partial x_i \partial x_j} + \sum_{i=1}^{n} b_i(\mathbf{r}) \frac{\partial f}{\partial x_i} + c(\mathbf{r}) f = g(\mathbf{r}), \tag{4}$$

where f is the solution, $\mathbf{r} = (x_1, \cdots, x_n)$ are the coordinates of the problem, and $a_{ij} = a_{ji}$ (to be elliptic, the quadratic form $\sum a_{ij}\xi_i\xi_j$ has to be positive-definite at every point $\mathbf{r}$).

In practice (that is, when analytic solutions are not available), efficient numerical procedures for solving elliptic PDEs are necessary. The PDE to be solved is cast (or *discretized* [23]) into a system of linear equations,

$$\mathbf{Ax} = \mathbf{b}, \tag{5}$$

where the square matrix $\mathbf{A}$ and the column vector $\mathbf{b}$ are obtained from the PDE, while $\mathbf{x}$ is related to its solution f (for example, $\mathbf{x}$ may represent the interpolation of f on some grid). The most popular ways of discretizing a PDE are the *finite difference method* and the *finite element method* (see, for example, Ref. [23]).

The success of both the finite difference and the finite element methods is a consequence of the fact that the resulting matrix $\mathbf{A}$ is *sparse*, that is, most of its elements are zero. The sparseness of $\mathbf{A}$ makes possible the solving of Eq. (5) by means of fast *iterative methods*, such as, to name a few, the *successive over-relaxation* (SOR) method, *multigrid* methods, or the *conjugate gradient* method (see Ref. [23] for a complete description).

A different approach, mentioned above regarding Eq. (2), is to use the method of Ritz [24] and express the solution of the PDE as an expansion of plane waves. When using Cartesian coordinates, the $\mathbf{A}$ matrix corresponding to Poisson's equation is diagonal and therefore the solution of Eq. (5) is trivial [1]. However, if we use adaptive coordinates instead of Cartesians, the resulting $\mathbf{A}$ matrix not only is not diagonal, it is not even sparse, as most of its elements are different from zero.

For these *dense* $\mathbf{A}$ matrices, the standard implementation of most iterative methods is very inefficient, both in computing

*jmpj@ua.es

time and memory requirements. It is possible, however, when solving Poisson's equation in adaptive coordinates, to compute very efficiently the product of $\mathbf{A}$ by some approximate solution $\tilde{\mathbf{x}}$ by means of the *fast Fourier transform* [25] (FFT), and this allows a fast solution of this problem [3] by means of the conjugate gradient method. For other iterative methods, such as SOR or multigrid, it is not obvious how we can take advantage of the FFT to implement fast versions of the respective algorithms.

This is the goal of the present work: to carry out efficient implementations of a variety of iterative methods by means of the FFT. The implementations will be tested on Poisson's equation in adaptive coordinates [21], but could be applied to any multidimensional elliptic PDE.

II. DISCRETIZATION OF PDEs

In this work we propose to solve a PDE by the method of Ritz [24] and express the solution as a linear combination of plane waves. In this section we explain how the PDE can be discretized, that is, replaced by a system of linear equations.

A. Calculus of variations

The solutions of PDEs can be obtained by means of the *calculus of variations* [24]. If our PDE is the Euler-Lagrange equation of some functional $J[\xi]$ depending on a trial function ξ, then the solution f will be the function ξ that minimizes J (more generally, the function ξ for which $J[\xi]$ is stationary). For example, it is easy to prove [26] that Poisson's equation in Cartesian coordinates, Eq. (1), is the Euler-Lagrange equation corresponding to the functional

$$J[\xi] = \frac{1}{8\pi}\int |\nabla\xi|^2\, d\mathbf{x} - \int \rho\xi\, d\mathbf{x}. \tag{6}$$

B. Expansions of cas functions

In the present work, we intended to express the solution of a PDE as a linear combination of plane waves. Considering, however, that the solution of PDEs such as Poisson's equation are real functions, while plane waves are complex functions, we prefer to use *cas functions* [27],

$$\operatorname{cas} x = \cos x + \sin x \tag{7}$$

("cas" stands for "cosine and sine"), which are the real counterparts of complex exponentials,

$$e^{ix} = \cos x + i \sin x. \tag{8}$$

For a multidimensional problem with n dimensions, we define the following basis functions,

$$\chi_{\mathbf{k}}(\mathbf{x}) = \frac{1}{(\sqrt{2\pi})^n} \operatorname{cas}(k_1 x_1)\cdots\operatorname{cas}(k_n x_n), \tag{9}$$

where $\mathbf{x} = (x_1, \cdots x_n)$, the integer components of the wave vector $\mathbf{k} \equiv (k_1, \cdots, k_n)$ are restricted as follows,

$$-\frac{N_j}{2} \leqslant k_j < \frac{N_j}{2}, \tag{10}$$

and N_j gives the size of the basis along the jth dimension, so that the total number of basis functions will be $N_1 \times \cdots \times N_n$.

With these basis functions we can express the trial function ξ as

$$\xi(\mathbf{x}) = \sum_{\mathbf{k}} c_{\mathbf{k}} \chi_{\mathbf{k}}(\mathbf{x}). \tag{11}$$

C. Fast Hartley transform

The FFT is an essential tool for handling plane-wave expansions. As cas functions can be considered "real" complex exponentials, it is not surprising that a similar tool exists for expansions of cas functions: the *fast Hartley transform* [27] (FHT). We will explain briefly its foundations, restricting ourselves to the one-dimensional case for simplicity.

Let f be the following expansion:

$$f(x) = \sum_{k=-N/2}^{N/2-1} c_k \operatorname{cas} kx, \tag{12}$$

where N is an even integer greater than 2. Let the set $\{f_j\}$ be the values that f takes at the abscissas of the *trapezoidal* rule,

$$f_j = f(2\pi j/N), \quad j = 0, \ldots, N-1 \tag{13}$$

(we assume that the argument of f is restricted to be in the interval $[0, 2\pi]$).

It is easy to prove that there is a *one-to-one* relation between the set of coefficients $\{c_k\}$ and the set of values $\{f_j\}$, that is, there is only a possible set of values $\{f_j\}$ for each set of coefficients $\{c_k\}$, and vice versa. A procedure for computing the set $\{c_k\}$ from $\{f_j\}$ is called a *direct Hartley transform*. Conversely, a procedure for obtaining the set $\{f_j\}$ from $\{c_k\}$ is called an *inverse Hartley transform*.

A naive implementation of the Hartley transform would have an $O(N^2)$ cost. For example, imagine that we compute a given $f_j = f(2\pi j/N)$ by direct application of Eq. (12). As the cost of computing each f_j will be proportional to N, and there are N values in the set $\{f_j\}$, the total cost will be proportional to N^2. Fortunately, there is a much more efficient way of computing the Hartley transform, the FHT.

The FHT is based upon the following equation:

$$f(x) = \cos(Nx/4) f_e(x) + \sin(Nx/4) f_o(-x), \tag{14}$$

where both f_e and f_o are themselves expansions of cas functions, but of length $N/2$, and are obtained, respectively, from the even and odd values of the set $\{f_j\}$,

$$f_e\left(\frac{2\pi j}{N/2}\right) = (-1)^j f_{2j}, \tag{15}$$

$$f_o\left(-\frac{2\pi j}{N/2} - \frac{2\pi}{N}\right) = (-1)^j f_{2j+1}. \tag{16}$$

The proof of Eq. (14) is not difficult. Considering the definition of cas functions given in Eq. (7), it is easy to arrive at the following relations,

$$2\operatorname{cas}\alpha\cos\beta = \operatorname{cas}(\alpha+\beta) + \operatorname{cas}(\alpha-\beta), \tag{17}$$

$$2\operatorname{cas}(-\alpha)\sin\beta = \operatorname{cas}(\alpha+\beta) - \operatorname{cas}(\alpha-\beta), \tag{18}$$

and, from these relations, it is simple to conclude that the right-hand side of Eq. (14) is a cas expansion of length N.

Furthermore, from the relations

$$\cos\left(\frac{N\{2\pi[2j]/N\}}{4}\right) = \cos(\pi j) = (-1)^j, \quad (19)$$

$$\sin\left(\frac{N\{2\pi[2j]/N\}}{4}\right) = \sin(\pi j) = 0, \quad (20)$$

$$\cos\left(\frac{N\{2\pi[2j+1]/N\}}{4}\right) = \cos(\pi j + \pi/2) = 0, \quad (21)$$

$$\sin\left(\frac{N\{2\pi[2j+1]/N\}}{4}\right) = \sin(\pi j + \pi/2) = (-1)^j, \quad (22)$$

we can see that the set values $\{f_j\}$ computed from both sides of Eq. (14) are equal.

It is trivial to prove, considering again Eqs. (17) and (18), that the functions f_e and f_o can be obtained from the coefficients of f,

$$f_e(x) = \sum_{k=-N/4}^{N/4-1} (c_{k+N/4} + c_{k-N/4}) \operatorname{cas}(kx), \quad (23)$$

$$f_o(x) = \sum_{k=-N/4}^{N/4-1} (c_{k+N/4} - c_{k-N/4}) \operatorname{cas}(kx). \quad (24)$$

Now it is easy to see how Eq. (14) can be used to implement a faster Hartley transform. Imagine that we want to compute the set of values $\{f_j\}$ from the set of coefficients $\{c_k\}$. Instead of computing the whole $\{f_j\}$ set from f, we can compute the even half of $\{f_j\}$ from f_e in Eq. (23), and the odd half from f_o in Eq. (24). As the length of both f_e and f_o is half that of f, the cost for each half would be $(N/2)^2 = N^2/4$, which will give a total cost of $2N^2/4 = N^2/2$. If we apply this procedure recursively to f_e and f_o, we will end up with a total cost proportional to $N \log N$.

In order to apply this algorithm recursively, N must be a power of 2. However, as there are FFT algorithms for N not a power of 2, one could adapt them to the FHT, and thus obtain a (hopefully) more efficient and flexible algorithm than the one presented above. For the moment, though, we will restrict ourselves to the simpler power of 2 case.

D. Discretization of PDEs for cas functions

Let us explain how to discretize Poisson's equation in Cartesian coordinates when the trial function ξ is expressed as in Eq. (11).

We begin by defining a square matrix $\mathbf{A}$ and a column matrix $\mathbf{b}$ with respective elements

$$(\mathbf{A})_{\mathbf{lm}} = \frac{1}{4\pi} \int \left(\frac{\partial \chi_\mathbf{l}}{\partial x}\frac{\partial \chi_\mathbf{m}}{\partial x} + \frac{\partial \chi_\mathbf{l}}{\partial y}\frac{\partial \chi_\mathbf{m}}{\partial y} + \frac{\partial \chi_\mathbf{l}}{\partial z}\frac{\partial \chi_\mathbf{m}}{\partial z} \right) d\mathbf{x} \quad (25)$$

and

$$(\mathbf{b})_\mathbf{l} = \int \rho \chi_\mathbf{l}\, d\mathbf{x}. \quad (26)$$

Next, we replace the expression of ξ given in Eq. (11) into Eq. (6), and arrive at the following equivalence:

$$J[\xi] = \frac{1}{2} \sum_\mathbf{l} \sum_\mathbf{m} c_\mathbf{l} (\mathbf{A})_{\mathbf{lm}} c_\mathbf{m} - \sum_\mathbf{l} (\mathbf{b})_\mathbf{l} c_\mathbf{l}. \quad (27)$$

Finally, to minimize $J[\xi]$, we equate its partial derivatives with respect to each $c_\mathbf{l}$ to zero and obtain the following system of linear equations:

$$\sum_\mathbf{m} (\mathbf{A})_{\mathbf{lm}} c_\mathbf{m} = (\mathbf{b})_\mathbf{l}. \quad (28)$$

In this way, we have cast the problem of solving Poisson's equation into the problem of solving a system of linear equations.

For Poisson's equation in Cartesian coordinates, the solving of this system of linear equation is trivial because the matrix $\mathbf{A}$ is diagonal. For other PDEs, or for this same Poisson's equation but in coordinates other than Cartesians, the solution is not so simple because $\mathbf{A}$, in general, will be a *dense* matrix. For example, if we use adaptive coordinates instead of Cartesians, the matrix elements of $\mathbf{A}$ take the following form:

$$(\mathbf{A})_{\mathbf{lm}} = 2 \int \left[\sum_{j=1}^{3} \sum_{k=1}^{3} Q_{jk} \frac{\partial \chi_\mathbf{l}}{\partial x_j} \frac{\partial \chi_\mathbf{m}}{\partial x_k} + \sum_{j=1}^{3} Q_j \left(\frac{\partial \chi_\mathbf{l}}{\partial x_j} \chi_\mathbf{m} + \chi_\mathbf{l} \frac{\partial \chi_\mathbf{m}}{\partial x_j} \right) + Q \chi_\mathbf{l} \chi_\mathbf{m} \right] d\mathbf{x}, \quad (29)$$

where $\mathbf{x} \equiv (x_1, x_2, x_3)$ represent the adaptive coordinates, and where Q, Q_j, and Q_{jk} are functions whose precise expression can be found in Ref. [21].

E. Evaluation of the matrix elements

We will discuss now the evaluation of the matrix elements of $\mathbf{A}$. For the sake of agility, we will use Dirac's notation, so that Eq. (29) can be rewritten as

$$(\mathbf{A})_{\mathbf{lm}} = 2 \sum_{j=1}^{3} \sum_{k=1}^{3} \left\langle \frac{\partial \chi_\mathbf{l}}{\partial x_j} \middle| Q_{jk} \middle| \frac{\partial \chi_\mathbf{m}}{\partial x_k} \right\rangle + 2 \sum_{j=1}^{3} \left(\left\langle \frac{\partial \chi_\mathbf{l}}{\partial x_j} \middle| Q_j \middle| \chi_\mathbf{m} \right\rangle + \left\langle \chi_\mathbf{l} \middle| Q_j \middle| \frac{\partial \chi_\mathbf{m}}{\partial x_j} \right\rangle \right) + 2 \langle \chi_\mathbf{l} | Q | \chi_\mathbf{m} \rangle. \quad (30)$$

The functions Q, Q_j, and Q_{jk} can be expanded by means of the $\chi_\mathbf{k}$ basis functions, in a similar way as was done in Eq. (11) for the trial solution ξ. For example, for Q, we will have

$$Q(\mathbf{x}) = \sum_\mathbf{k} q_\mathbf{k} \chi_\mathbf{k}(\mathbf{x}). \quad (31)$$

At first sight, it may seem that in order to yield exact results for $(\mathbf{A})_{\mathbf{lm}}$, these expansions have to be of infinite length. However, as the derivative of a cas function is another cas function, and the product of two cas functions is a short linear combination of cas functions (recall that cas functions are nothing more than "real" complex exponentials), it turns out that an expansion twice as long (along each dimension) as that of ξ suffices to get exact results.

We will use the notation $\bar{Q}$, $\bar{Q}_j$, and $\bar{Q}_{jk}$ to refer to these truncated counterparts of Q, Q_j, and Q_{jk}, and, as the truncated

expansions also yield exact results, we will have

$$(\mathbf{A})_{\mathbf{lm}} = 2\sum_{j=1}^{3}\sum_{k=1}^{3}\left\langle \frac{\partial \chi_{\mathbf{l}}}{\partial x_j}\,\middle|\,\bar{Q}_{jk}\,\middle|\,\frac{\partial \chi_{\mathbf{m}}}{\partial x_k}\right\rangle + 2\sum_{j=1}^{3}\left(\left\langle \frac{\partial \chi_{\mathbf{l}}}{\partial x_j}\,\middle|\,\bar{Q}_j\,|\,\chi_{\mathbf{m}}\right\rangle + \left\langle \chi_{\mathbf{l}}\,|\,\bar{Q}_j\,|\,\frac{\partial \chi_{\mathbf{m}}}{\partial x_j}\right\rangle\right) + 2\langle \chi_{\mathbf{l}}\,|\,\bar{Q}\,|\,\chi_{\mathbf{m}}\rangle. \tag{32}$$

The integrals in this expression can be computed accurately and efficiently by numerical integration combined with the FHT (see Ref. [16] for the details).

It is possible to transforms some of the integrals in Eq. (32) by means of integration by parts. For example, it is easy to see that

$$\left\langle \frac{\partial \chi_{\mathbf{l}}}{\partial x_j}\,\middle|\,\bar{Q}_j\,|\chi_{\mathbf{m}}\right\rangle = -\langle \chi_{\mathbf{l}}|\,\frac{\partial \bar{Q}_j}{\partial x_j}|\,\chi_{\mathbf{m}}\rangle - \langle \chi_{\mathbf{l}}|\,\bar{Q}_j\,\middle|\,\frac{\partial \chi_{\mathbf{m}}}{\partial x_j}\right\rangle. \tag{33}$$

In a similar manner, it is possible to rewrite Eq. (32) as

$$(\mathbf{A})_{\mathbf{lm}} = \langle \chi_{\mathbf{l}}\,|\,O|\,\chi_{\mathbf{m}}\rangle + \sum_{j=1}^{3}\langle \chi_{\mathbf{l}}\,|\,O_j\,\middle|\,\frac{\partial \chi_{\mathbf{m}}}{\partial x_j}\right\rangle + \sum_{j=1}^{3}\sum_{k=1}^{j}\langle \chi_{\mathbf{l}}\,|\,O_{jk}\,\middle|\,\frac{\partial^2 \chi_{\mathbf{m}}}{\partial x_j \partial x_k}\right\rangle, \tag{34}$$

where the explicit form of O, O_j, and O_{jk} can be easily worked out from $\bar{Q}$, $\bar{Q}_j$, and $\bar{Q}_{jk}$ and their derivatives.

F. Partial differential operators

Let f and g be any two functions expressed as linear combinations analogous to the one used for ξ in Eq. (11). Equation (34) suggests the introduction of a *partial differential operator* $\hat{O}$ that acts on f to produce g,

$$g = \hat{O}\,f. \tag{35}$$

The effect of $\hat{O}$ over any f can be obtained as follows.

(1) The following temporary function g_0 is computed:

$$g_0 = Of + \sum_{j=1}^{3} O_j \frac{\partial f}{\partial x_j} + \sum_{j=1}^{3}\sum_{k=1}^{j} O_{jk}\frac{\partial^2 f}{\partial x_j \partial x_k}. \tag{36}$$

Note that the derivatives of f, as well as O, O_j, and O_{jk}, are all linear combinations of cas functions, and therefore g_0 can be efficiently evaluated via convolution and FHT with a cost of $O(N \log N)$, N being the number of cas functions used.

(2) This g_0 is a linear combination of cas functions, though longer than required in Eq. (11). Therefore, g_0 should be truncated to produce the desired result g.

According to this definition, it is obvious that the matrix elements of $\mathbf{A}$ can be expressed with the help of $\hat{O}$ as

$$(\mathbf{A})_{\mathbf{lm}} = \langle \chi_{\mathbf{l}}|\hat{O}|\chi_{\mathbf{m}}\rangle\,. \tag{37}$$

As a consequence, the system of linear equations given in Eq. (28) can be rewritten as

$$\hat{O}\,f = g, \tag{38}$$

where f is the solution and g is related to column matrix $\mathbf{b}$ as follows:

$$g(\mathbf{x}) = \sum_{\mathbf{k}} (\mathbf{b})_{\mathbf{k}}\,\chi_{\mathbf{k}}(\mathbf{x}). \tag{39}$$

G. Bisection of operators

We have seen in Eq. (14) that a cas expansion can be split into even and odd halves. We will extend this idea to the partial differential operators introduced above. For simplicity, we will restrict ourselves to the one-dimensional case, where a differential operator has the following expression:

$$\hat{O}(x) = O_0(x) + O_1(x)\frac{d}{dx} + O_2(x)\frac{d^2}{dx^2}. \tag{40}$$

Let us have two cas expansions f and g, both of length N with N even and higher than 2. According to Eq. (14), we can split f and g as follows:

$$f(x) = \cos(Nx/4) f_e(x) + \sin(Nx/4) f_o(-x), \tag{41}$$

$$g(x) = \cos(Nx/4) g_e(x) + \sin(Nx/4) g_o(-x). \tag{42}$$

It is straightforward, though tedious, to bisect the operator $\hat{O}$ into even and odd components,

$$\langle f|\hat{O}|g\rangle = \langle f_e(x)|\hat{O}_{ee}|g_e(x)\rangle + \langle f_e(x)|\hat{O}_{eo}|g_o(-x)\rangle + \langle f_o(x)|\hat{O}_{oo}|g_o(x)\rangle + \langle f_o(x)|\hat{O}_{oe}|g_e(-x)\rangle, \tag{43}$$

with $\hat{O}_{ee}$, $\hat{O}_{eo}$, $\hat{O}_{oo}$, and $\hat{O}_{oe}$ defined as follows:

$$\hat{O}_{ee} = \left[O_0^{cc}(x) - \frac{N}{4}O_1^{cs}(x) - \frac{N^2}{16}O_2^{cc}(x)\right] + \left[O_1^{cc}(x) - \frac{N}{2}O_2^{cs}(x)\right]\frac{d}{dx} + O_2^{cc}(x)\frac{d^2}{dx^2}, \tag{44}$$

$$\hat{O}_{eo} = \left[O_0^{cs}(x) + \frac{N}{4}O_1^{cc}(x) - \frac{N^2}{16}O_2^{cs}(x)\right] + \left[O_1^{cs}(x) + \frac{N}{2}O_2^{cc}(x)\right]\frac{d}{dx} + O_2^{cs}(x)\frac{d^2}{dx^2}, \tag{45}$$

$$\hat{O}_{oo} = \left[O_0^{ss}(x) + \frac{N}{4}O_1^{sc}(x) - \frac{N^2}{16}O_2^{ss}(x)\right] - \left[O_1^{ss}(x) + \frac{N}{2}O_2^{sc}(x)\right]\frac{d}{dx} + O_2^{ss}(x)\frac{d^2}{dx^2}, \tag{46}$$

$$\hat{O}_{oe} = \left[O_0^{sc}(x) - \frac{N}{4}O_1^{ss}(x) - \frac{N^2}{16}O_2^{sc}(x)\right] - \left[O_1^{sc}(x) - \frac{N}{2}O_2^{ss}(x)\right]\frac{d}{dx} + O_2^{sc}(x)\frac{d^2}{dx^2}, \tag{47}$$

and where

$$O_i^{cc}(x) = \cos^2(Nx/4) O_i(x), \tag{48}$$

$$O_i^{ss}(x) = \sin^2(Nx/4) O_i(-x), \tag{49}$$

$$O_i^{cs}(x) = \cos(Nx/4)\sin(Nx/4) O_i(x), \tag{50}$$

$$O_i^{sc}(x) = -\sin(Nx/4)\cos(Nx/4) O_i(-x). \tag{51}$$

Note that the lengths of the cas expansions in $\hat{O}_{ee}$, $\hat{O}_{eo}$, $\hat{O}_{oo}$, and $\hat{O}_{oe}$, after truncation, are half the length of those in $\hat{O}$.

It is straightforward to arrive at the following relation:

$$\langle f|g\rangle = \tfrac{1}{2}\langle f_e|g_e\rangle + \tfrac{1}{2}\langle f_o|g_o\rangle, \tag{52}$$

and from this last expression and Eq. (43) we can easily arrive at

$$\hat{O}_{ee} f_e(x) + \hat{O}_{eo} f_o(-x) = \tfrac{1}{2} g_e(x), \tag{53}$$

$$\hat{O}_{oe} f_e(-x) + \hat{O}_{oo} f_o(x) = \tfrac{1}{2} g_o(x). \tag{54}$$

Comparing these two equations with Eq. (38), we see that now we have two half-equations, one for f_e and the other for f_o. Note that these equations are coupled, as both f_e and f_o appear in both sides of them. We will explain next how to solve these equations by means of iterative methods.

III. ITERATIVE METHODS

After the discretization of a PDE (see Sec. II D), we are left with a system of linear equations to solve

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{55}$$

The matrix $\mathbf{A}$ is dense, so that an explicit computation and storage of its elements is not feasible for large systems, thus making very expensive, or even impossible, the use of direct methods for large problems. Suppose, for example, that we try to solve our system via a Cholesky decomposition [28]. If N is the number of cas functions used, then the cost of this decomposition will be $O(N^3)$ in CPU time and $O(N^2)$ in memory, which will become quickly out of the question as N grows.

In this section, we will discuss better alternatives, based on iterative methods, that do not require the explicit computation and storage of matrix $\mathbf{A}$.

A. Stationary iterative methods

Stationary (also known as *relaxation* or *basic*) iterative methods [23,29] include procedures for solving systems of linear equation such as Jacobi, Gauss-Seidel (GS), successive over-relaxation (SOR), or symmetric SOR (SSOR) methods.

These methods work by decomposing the matrix $\mathbf{A}$ into three components: diagonal ($\mathbf{D}$), strictly lower triangular ($\mathbf{L}$, with the elements of $\mathbf{A}$ below the diagonal), and strictly upper diagonal ($\mathbf{U}$, with the elements of $\mathbf{A}$ above the diagonal),

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}. \tag{56}$$

With the help of these matrices, the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ is rearranged, for example, as

$$(\mathbf{D} + \mathbf{L})\mathbf{x} = \mathbf{b} - \mathbf{U}\mathbf{x}, \tag{57}$$

and then the solution is obtained iteratively: a sequence of approximate solutions $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(m)}$ is generated so that $\mathbf{x}^{(m+1)}$ is computed from $\mathbf{x}^{(m)}$ according to the relation

$$(\mathbf{D} + \mathbf{L})\mathbf{x}^{(m+1)} = \mathbf{b} - \mathbf{U}\mathbf{x}^{(m)}. \tag{58}$$

As $\mathbf{D} + \mathbf{L}$ is a lower diagonal matrix, this system is trivially solved.

The procedure just explained is known as the Gauss-Seidel method. The sequence of solutions $\mathbf{x}^{(m)}$ is guaranteed [29] to converge to the true solution if $\mathbf{A}$ is a symmetric positive-definite matrix.

If implemented as explained above, the GS method is only practical if $\mathbf{A}$ is a sparse matrix. For dense matrices, the cost of solving Eq. (58) will be $O(N^3)$ in CPU time and $O(N^2)$ in memory, and therefore, as many iterations have to be carried out, the overall performance would be much worse than that of a direct approach.

Nevertheless, we propose an alternative implementation of the GS method for the problem at hand, implementation that does not require the storage of the matrices $\mathbf{L}$ and $\mathbf{U}$ in memory, and that allows an efficient solution of Eq. (58) via the FHT. We give the details next.

Recall that we are trying to solve a PDE that has been rewritten as $\hat{O} f = g$ in Eq. (38), where f is the solution. We begin by casting Eqs. (53) and (54) in matrix form,

$$\begin{bmatrix} \hat{O}_{ee} & \hat{O}_{eo}\hat{M} \\ \hat{O}_{oe}\hat{M} & \hat{O}_{oo} \end{bmatrix} \begin{bmatrix} f_e \\ f_o \end{bmatrix} = \frac{1}{2} \begin{bmatrix} g_e \\ g_o \end{bmatrix}, \tag{59}$$

where the effect of $\hat{M}$ over any cas expansion h is the following:

$$\hat{M} h(x) = h(-x). \tag{60}$$

Next, inspired by the decomposition of $\mathbf{A}$ in Eq. (56), we apply a similar one to the matrix in Eq. (59):

$$\begin{bmatrix} \hat{O}_{ee} & \hat{O}_{eo}\hat{M} \\ \hat{O}_{oe}\hat{M} & \hat{O}_{oo} \end{bmatrix} = \overbrace{\begin{bmatrix} \hat{O}_{ee} & 0 \\ 0 & \hat{O}_{oo} \end{bmatrix}}^{\mathbf{D}} + \overbrace{\begin{bmatrix} 0 & 0 \\ \hat{O}_{oe}\hat{M} & 0 \end{bmatrix}}^{\mathbf{L}} + \overbrace{\begin{bmatrix} 0 & \hat{O}_{eo}\hat{M} \\ 0 & 0 \end{bmatrix}}^{\mathbf{U}}. \tag{61}$$

Finally, from these $\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$ matrices, the GS iterations in Eq. (58) can be written as a system of two equations, one for the even part f_e, and the other for the odd part f_o:

$$\hat{O}_{ee} f_e^{(m+1)}(x) = \tfrac{1}{2} g_e(x) - \hat{O}_{eo} f_o^{(m)}(-x), \tag{62}$$

$$\hat{O}_{oo} f_o^{(m+1)}(x) = \tfrac{1}{2} g_o(x) - \hat{O}_{oe} f_e^{(m+1)}(-x). \tag{63}$$

Equations (62) and (63) (with similar equations for other stationary methods such as SOR or SSOR) constitute the cornerstone of the present work. We will explain in detail their importance:

(1) The equation $\hat{O} f = g$ is solved iteratively, with a sequence of approximate solutions $f^{(0)}, \ldots f^{(m)}$ that converges to the exact solution f. The functions $f_e^{(m)}$ and $f_o^{(m)}$ represent, according to Eq. (14), the even and odd halves of $f^{(m)}$.

(2) The original equation $\hat{O} f = g$ has N unknowns, while Eqs. (62) and (63) have $N/2$ unknowns each, and therefore are easier to solve. Instead of solving these two equations by a direct method, we can proceed recursively by applying successive bisections, so that at the next level of bisection we will have four equations of $N/4$ unknowns each, then eight

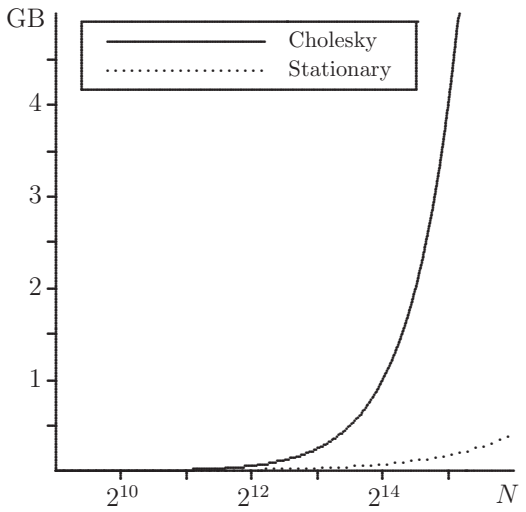

FIG. 1. Memory requirement (in Gigabytes, GB) for the Cholesky decomposition and the stationary iterative methods when using double-precision arithmetic. N is the total number of 3D cas functions (or unknowns). For the stationary iterative methods, a $8 \times 8 \times 8$ grid is used for the bottom level.

equations of $N/8$ unknowns, and so on. At the *bottom level*, when we consider that no more bisections are advisable, we can solve the equations by a direct method (we will use Cholesky decomposition), because, at this bottom level, the number of unknowns for each equation is small enough to make direct methods practical.

(3) $\hat{O}_{eo}$ and $\hat{O}_{oe}$ should be stored for all levels but the bottom one, with memory requirements of the order of $O(N \log N)$. The operators $\hat{O}_{ee}$ and $\hat{O}_{oo}$ do not have to be stored because they are not directly used but recursively forwarded to the next level and then bisected. Finally, at the bottom level, we do have to store all the Cholesky decompositions, which needs an amount of memory of the order of $O(N)$. Overall, the memory requirements of stationary methods implemented as proposed here will be of the order of $O(N \log N)$. We compare in Fig. 1 the memory requirements of a stationary iterative method such as the GS method and the corresponding requirements for a Cholesky decomposition. We see that, even if the matrix $\mathbf{A}$ is not sparse, the memory needs of our implementation of the GS method are much smaller than those of a Cholesky decomposition. Note that $\hat{O}_{eo}$ and $\hat{O}_{oe}$ are precomputed and stored at the beginning of the calculation, and then are used for all the subsequent iterations of the GS method.

(4) The CPU time required for each iteration of our GS method is dominated by the evaluation of $\hat{O}_{eo} f_o^{(m)}(-x)$ and $\hat{O}_{oe} f_e^{(m+1)}(-x)$. This evaluation can be efficiently done via FHT and convolution and has a cost of $O(N \log N)$ for each recursion level. As the number of levels is proportional to $\log N$, we will have a total cost of $O(N \log^2 N)$, which is much better than the $O(N^2)$ cost of a GS iteration for dense matrices.

The procedure just explained can be trivially generalized for other stationary iterative methods. We will consider briefly the SOR and SSOR methods.

For the SOR method, Eq. (58) should be replaced by the following counterpart:

$$(\mathbf{D} + \omega \mathbf{L})\mathbf{x}^{(m+1)} = \omega \mathbf{b} - [\omega \mathbf{U} + (\omega - 1)\mathbf{D}]\mathbf{x}^{(m)}, \tag{64}$$

where ω is the *relaxation factor*. The SOR method is guaranteed [29] to converge if $\mathbf{A}$ is symmetric and positive-definite and ω has a value between 0 and 2 (note that for $\omega = 1$ SOR reverts to the GS method). With our definition of matrices $\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$ in Eq. (61), we get

$$\hat{O}_{ee}\left[\frac{1}{\omega} f_e^{(m+1)}(x) + \frac{\omega - 1}{\omega} f_e^{(m)}(x)\right] = \frac{1}{2} g_e(x) - \hat{O}_{eo} f_o^{(m)}(-x), \tag{65}$$

$$\hat{O}_{oo}\left[\frac{1}{\omega} f_o^{(m+1)}(x) + \frac{\omega - 1}{\omega} f_o^{(m)}(x)\right] = \frac{1}{2} g_o(x) - \hat{O}_{oe} f_e^{(m+1)}(-x), \tag{66}$$

which suggest the following strategy:

(1) Solve

$$\hat{O}_{ee} \tilde{f}_e^{(m+1)}(x) = \tfrac{1}{2} g_e(x) - \hat{O}_{eo} f_o^{(m)}(-x), \tag{67}$$

for $\tilde{f}_e^{(m+1)}(x)$ with

$$\tilde{f}_e^{(m+1)}(x) = \frac{1}{\omega} f_e^{(m+1)}(x) + \frac{\omega - 1}{\omega} f_e^{(m)}(x). \tag{68}$$

(2) Compute $f_e^{(m+1)}(x)$ as follows:

$$f_e^{(m+1)}(x) = \omega \tilde{f}_e^{(m+1)}(x) - (\omega - 1) f_e^{(m)}(x). \tag{69}$$

This should be done only at the bottom level, to avoid multiplying by ω or $\omega - 1$ more than once.

(3) These steps should be repeated in a similar way for Eq. (66).

For the SSOR method, we have

$$(\mathbf{D} + \omega \mathbf{L})\mathbf{x}^{(m+1/2)} = \omega \mathbf{b} - [\omega \mathbf{U} + (\omega - 1)\mathbf{D}]\mathbf{x}^{(m)}, \tag{70}$$

$$(\mathbf{D} + \omega \mathbf{U})\mathbf{x}^{(m+1)} = \omega \mathbf{b} - [\omega \mathbf{L} + (\omega - 1)\mathbf{D}]\mathbf{x}^{(m+1/2)}, \tag{71}$$

which is equivalent to a SOR forward step followed by a SOR backward step. For $\omega = 1$, the SSOR method is just the symmetric Gauss-Seidel (SGS) method.

B. Krylov subspace methods

Krylov subspace methods [23] are a different kind of iterative methods for solving a system of linear equations. Given the system $\mathbf{Ax} = \mathbf{b}$ and a vector $\mathbf{r}_0$, Krylov methods work by generating the sequence $\mathbf{r}_0$, $\mathbf{Ar}_0$, $\mathbf{A}^2\mathbf{r}_0$, . . ., and then approximating the solution $\mathbf{x}$ as a linear combination of these vectors. The *Krylov subspace* is the subspace spanned by the vectors in the sequence,

$$\mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, \mathbf{Ar}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{m-1}\mathbf{r}_0\}, \tag{72}$$

which means that the approximated solution belongs to $\mathcal{K}_m(\mathbf{A}, \mathbf{r}_0)$. The vector $\mathbf{r}_0$ is defined as $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$, where $\mathbf{x}_0$ is some initial guess to the solution.

As we have explained in Sec. II F, the PDE that we are trying to solve is rewritten in operator form as $\hat{O} f = g$, so that given

an initial guess f_0 to the solution f, we have to generate the sequence $r_0 = g - \hat{O} f_0$, $\hat{O} r_0$, $\hat{O}(\hat{O} r_0), \ldots$ in order to build up the Krylov subspace. This task reduces itself to applying the operator $\hat{O}$ to the previous vector in the sequence, and can be efficiently accomplished via FHT and convolution (see Sec. II F).

The *conjugate gradient* (CG) method [23] is one of the most popular Krylov methods. It can be used when the matrix $\mathbf{A}$ is symmetric and positive-definite, as is the case for Poisson's equation. The corresponding algorithm is neat and simple and has an important advantage: there is no need to store the whole sequence of vectors that span the Krylov subspace, it suffices to store a total of four vectors.

The efficiency of Krylov methods can be improved by using *preconditioning*, which replaces the original system of linear equations to be solved by another one with the same solution but (hopefully) easier to solve. Given a suitable preconditioning matrix or *preconditioner* $\mathbf{M}$ we can transform the system $\mathbf{Ax} = \mathbf{b}$ into an alternative system,

$$(\mathbf{AM}^{-1})\mathbf{y} = \mathbf{b}, \tag{73}$$

solve it for $\mathbf{y}$, and then obtain $\mathbf{x}$ from $\mathbf{y}$,

$$\mathbf{x} = \mathbf{M}^{-1}\mathbf{y}. \tag{74}$$

This particular scheme is called *right preconditioning*.

Once a preconditioner has been selected, the Krylov subspace for the preconditioned system is the following:

$$\mathcal{K}_m(\mathbf{AM}^{-1}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, \mathbf{AM}^{-1}\mathbf{r}_0, (\mathbf{AM}^{-1})^2 \times \mathbf{r}_0, \ldots, (\mathbf{AM}^{-1})^{m-1}\mathbf{r}_0\}. \tag{75}$$

There are many choices [23] for the preconditioner $\mathbf{M}$. One possibility, which is particularly simple for the present work, is to use the stationary iterative methods studied above as preconditioners. We will discuss its implementation next.

Consider the GS iteration in Eq. (58) and the following choice for the preconditioner:

$$\mathbf{M} = \mathbf{D} + \mathbf{L}. \tag{76}$$

Then the GS iteration can be written as

$$\mathbf{Mx}^{(m+1)} = \mathbf{b} - \mathbf{Ux}^m, \tag{77}$$

or, alternatively, as

$$\mathbf{x}^{(m+1)} = \mathbf{M}^{-1}\mathbf{b} - \mathbf{M}^{-1}\mathbf{Ux}^m. \tag{78}$$

It is easy to generate the Krylov subspace for this choice of preconditioner:

(1) First, note that it is not necessary to store the whole $\mathbf{M}$ matrix in memory. According to Eq. (75), it suffices that, given a vector $\mathbf{v}$, we are able to compute $\mathbf{M}^{-1}\mathbf{v}$.

(2) In order to obtain $\mathbf{M}^{-1}\mathbf{v}$, we could use the same computer code that implements a GS iteration: in Eq. (78), we just make $\mathbf{x}^m$ equal to zero and set $\mathbf{b}$ to $\mathbf{v}$. Then the result $\mathbf{x}^{(m+1)}$ returned by the program will be equal to $\mathbf{M}^{-1}\mathbf{v}$.

For SOR, SSOR, or any other stationary iterative method, the procedure is analogous, we just have to use the computer code that implements the respective iteration.

The conjugate gradient (CG) method mentioned above can be extended to deal with preconditioners in what is called the *preconditioned conjugate gradient* (PCG) method [23]. However, as the CG does not converge for nonsymmetric problems, it turns out that both the matrix $\mathbf{A}$ and the preconditioner $\mathbf{M}$ have to be symmetric for the PCG method to converge. This precludes the use of GS or SOR preconditioners, as they are not symmetric. The SSOR and SGS preconditioners are symmetric and therefore allowed for PCG.

If one is interested in nonsymmetric preconditioners, one alternative to PCG could be the *generalized minimal residual* (GMRES) method [23], which is a robust Krylov subspace method that also works for nonsymmetric problems. A downside of the GMRES method is that the whole sequence of vectors spanning the Krylov subspace has to be kept in memory, which could be prohibitive if a large number of iterations is needed (note that there are variants [23] of the GMRES method that try to remedy this problem, such as the *restarted GMRES method*).

C. Multigrid methods

Multigrid methods [30] are among the most powerful methods available for solving PDEs. Let us first explain what we mean by a *grid* in the present context.

Recall from Sec. II C that a cas expansion f of length N can be expressed as the set of coefficients $\{c_k\}$ of the cas functions, or as the set of values $\{f_j\}$ that f takes at the abscissas of a N-point trapezoidal rule. We will refer to this set of N abscissas as a *grid* of N points. For a 3D problem, we will extend this definition and speak, for instance, of a $N \times N \times N$ grid.

A multigrid method uses more than a grid to solve a problem. For example, it can use a *fine* grid $N \times N \times N$, and a *coarse* grid $N/2 \times N/2 \times N/2$, or even coarser grids $N/4 \times N/4 \times N/4, \ldots$

A crucial ingredient of multigrid methods is what is called *intergrid transfer*. Suppose that we have a function f^N represented in a fine grid and we need also its representation $f^{N/2}$ in a coarse grid. The process of obtaining $f^{N/2}$ from f^N (fine to coarse) is called *restriction*, while the opposite operation from $f^{N/2}$ to f^N (coarse to fine) is called *interpolation* or *prolongation*.

Both restriction and interpolation are very easy to carry out in our current representation of functions as expansions of cas functions. Let the fine-grid function f^N be

$$f^N(x) = \sum_{k=-N/2}^{N/2-1} c_k \operatorname{cas} kx \tag{79}$$

(for simplicity, we will assume 1D grids). Then the restriction operation is just removing half the terms (those with higher $|k|$) from the previous summation,

$$f^{N/2}(x) = \sum_{k=-N/4}^{N/4-1} c_k \operatorname{cas} kx. \tag{80}$$

The interpolation operation from $f^{N/2}$ to f^N is also trivial: we copy all the coefficients in $f^{N/2}$ to f^N, and then set the remaining coefficients of f^N to zero.

In this work we have rewritten the equation to be solved in operator form, $\hat{O} f = g$, and therefore we need the operator $\hat{O}$ represented in all the grids that we are going to use

$(\hat{O}^N, \hat{O}^{N/2}, \ldots)$. These representations are easy to obtain via the restriction operation described above: we set $\hat{O}^N = \hat{O}$, and then we get $\hat{O}^{N/2}$ by restriction of $\hat{O}^N$, $\hat{O}^{N/4}$ by restriction of $\hat{O}^{N/2}$, and so on.

The multigrid algorithm is recursive in nature. We will give a short description of the steps required to perform a single iteration or *cycle*.

(1) On the current $N \times \cdots \times N$ grid, do ν_1 relaxation steps on $\hat{O}^N f^N = g^N$ with a given initial guess f^N (a "relaxation step" is a single iteration of an stationary iterative method such as the GS method).

(2) If we are on the coarsest grid, go to step 7.

(3) Compute $g^{N/2}$ by restriction of the residual $g^N - \hat{O}^N f^N$.

(4) Set $f^{N/2}$ to zero.

(5) Recursion. Go to step 1, but setting N to $N/2$. The recursion will produce an updated result for $f^{N/2}$.

(6) After the recursion, correct f^N by adding to it the interpolation of $f^{N/2}$ to the grid $N \times \cdots \times N$.

(7) Do ν_2 relaxation steps on $\hat{O}^N f^N = g^N$ with initial guest f^N.

The algorithm just described constitutes what is called a *V-cycle*. A related cycle, the *W-cycle*, is obtained if step 5 is repeated twice. In this work, we will restrict ourselves to the V-cycle, as the W-cycle does not produce better results.

Finally, we will point out that multigrid methods can be accelerated by using them as preconditioners of Krylov subspace methods. A V-cycle will be symmetric if the relaxation steps are performed by a symmetric stationary iteration such as SSOR or SGS and $\nu_1 = \nu_2$. In such a case, we can use this V-cycle as a preconditioner for the PCG. For nonsymmetric V-cycles (when $\nu_1 \neq \nu_2$ or the stationary iteration is not symmetric), we should use the GMRES method instead.

IV. BENCHMARKING

In this section we will offer some benchmarking of the methods described above. The results are presented in Table I and Fig. 2. In both cases, the model problem studied is Poisson's equation rewritten in adaptive coordinates, as explained in detail in Ref. [21], and applied to the *promolecule density* (spherical atomic densities centered at the nuclei of the molecule) of the H_3^+ molecule in its equilibrium geometry [31]. The methods have also been applied to a nonplanar, methane-like molecule (4 hydrogen atoms located at the vertices of a tetrahedron, plus another hydrogen atom at its center), but the results, being very similar to those obtained with the H_3^+ molecule, are not reported here.

We will use the *residual norm* as a measure of the error of the solutions obtained by the different methods. If, in operator form, the equation to be solved is $\hat{O} f = g$, and ξ is an approximated solution, then the residual norm ϵ is defined as

$$\epsilon = \sqrt{\langle \hat{O}\xi - g | \hat{O}\xi - g \rangle}. \tag{81}$$

We show in Fig. 2 the error (residual norm) as a function of the number of iterations for the GS method, the SOR method (with the value of the relaxation parameter optimized to $\omega = 1.89$), and the preconditioned conjugate gradient method with a symmetric Gauss-Seidel preconditioner (PCG-SGS). We conclude the following:

(1) Although the GS method is guaranteed [29] to converge to the true solution for symmetric positive-definite matrices, we see that the convergence is so slow as to make the method useless for the present problem. Also, the slight increase in the error at the beginning of the iteration is a consequence of the use, as an estimate of the error, the residual norm instead of the error norm, $\sqrt{\langle \xi - f | \xi - f \rangle}$, which, although will decrease monotonically [29], is not as easy to calculate as the residual norm because we do not know the exact solution f beforehand.

(2) The error of the SOR and PCG-SGS methods decreases steadily as the number of iteration increases, so that, if we keep ourselves within machine accuracy, a given target

TABLE I. CPU time (in seconds) required to solve the model problem by different methods. In all cases, the error (residual norm) is smaller than 10^{-9}. Two 3D grids of cas functions are used: $16 \times 16 \times 16$ and $32 \times 32 \times 32$, with a $8 \times 8 \times 8$ grid for the bottom level. The Cholesky entry marked by an asterisk is not directly computed but, due to the huge amount of memory that would be required, estimated by extrapolation. The multigrid preconditioner (MG) uses two grids ($8 \times 8 \times 8$ and $16 \times 16 \times 16$) for the $16 \times 16 \times 16$ column and three grids ($8 \times 8 \times 8$, $16 \times 16 \times 16$, and $32 \times 32 \times 32$) for the $32 \times 32 \times 32$ column.

	$16 \times 16 \times 16$	$32 \times 32 \times 32$
Cholesky	47.8	11115.4*
CG	63.9	4626.0
PCG-SGS	56.3	3399.4
GMRES-GS	36.8	1949.3
PCG-MG[SGS, $\nu_1 = \nu_2 = 1$]	29.2	1661.6
GMRES-MG[GS, $\nu_1 = 1$, $\nu_2 = 0$]	19.7	997.6
GMRES-MG[GS, $\nu_1 = \nu_2 = 1$]	20.2	1059.7

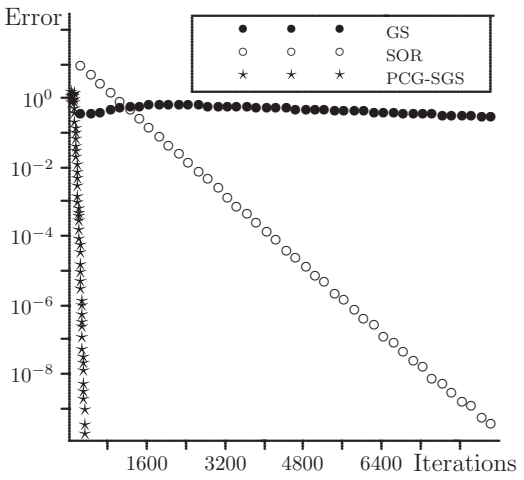

FIG. 2. Error (residual norm) as a function of the number of iterations for the GS method, the SOR method (with $\omega = 1.89$), and the PCG-SGS method. A 3D grid of cas functions of size $16 \times 16 \times 16$ is used in all cases, with a $8 \times 8 \times 8$ grid for the bottom level.

accuracy could be reached provided that enough iterations are carried out.

(3) Regarding the number of iterations, the PCG-SGS method clearly outperforms the other two methods. However, as the SGS preconditioner requires two half-iterations for each iteration, one may wonder how this would affect the corresponding CPU time. We have found that, on average, each complete PCG-SGS iteration takes about 72% more time than a GS or SOR iteration. Therefore, even accounting for these two half-iterations, the PCG-SGS method is by far the best method of the three.

The CPU time needed to solve the model problem for different methods is shown in Table I. In this table, "Cholesky" refers to the direct solution of the problem via Cholesky decomposition, "CG" is the conjugate gradient method with no preconditioner, while "PCG-*" and "GMRES-*" denote the preconditioned CG or the preconditioned GMRES methods with a given preconditioner "*," where "*" may stand for GS, SGS, or multigrid (MG). For this last preconditioner, we indicate in square brackets which stationary iteration is used for the relaxation step (SG or SGS), and the number of relaxation steps carried out before (ν_1) and after (ν_2) the multigrid recursion. We present some conclusions on this table:

(1) The cost of Cholesky scales as $O(N^3)$, which is worse than the scaling of the other methods, and causes this method to perform poorly for large grids. The stationary iterative methods have a $O(N \log^2 N)$ cost per iteration, but one should keep in mind that the number of iterations required to reach a given precision increases as N increases.

(2) As expected, the conjugate gradient method with no preconditioner (CG) is worse than the other Krylov subspace methods with preconditioner (PCG-* and GMRES-*).

(3) The cost of GMRES-GS is lower than that of PCG-SGS, so that we can conclude that the GS preconditioner is better than the SGS one (note that GS is not a symmetric preconditioner and therefore cannot be used with PCG). We would like to point out that although we have tested the SSOR preconditioner with different values of the relaxation parameter ω, it turns out that the optimum value is $\omega = 1$, which makes SSOR equivalent to SGS.

(4) Regarding multigrid methods accelerated via PCG or GMRES, we conclude, as in the previous point, that GS relaxation performs better than SGS relaxation. The fastest methods are GMRES-MG[GS, $\nu_1 = 1$, $\nu_2 = 0$] and GMRES-MG[GS, $\nu_1 = \nu_2 = 1$]. We caution, however, that GMRES has to store in memory the whole sequence of vectors spanning the Krylov subspace (one vector per iteration), so that, if not enough memory is available, we should use the PCG-MG[SGS, $\nu_1 = \nu_2 = 1$] method instead. Note also that GMRES-MG[GS, $\nu_1 = 1$, $\nu_2 = 0$] requires more iterations than GMRES-MG[GS, $\nu_1 = \nu_2 = 1$] (for example, for the $32 \times 32 \times 32$ grid, the first method needs 417 iterations, while the second only needs 265 iterations), so that, as the cost in CPU of both methods is similar, it is better to use GMRES-MG[GS, $\nu_1 = \nu_2 = 1$] when memory is scarce.

V. CONCLUSIONS

In this work, we have proposed a new method for solving arbitrary elliptic partial differential equations. We outline its main features and give some conclusions:

(1) The approach is based upon the method of Ritz, a variational method for solving PDEs. The solution is written as a linear combination of plane waves, so that the coefficients of the plane waves are found by variational minimization.

(2) This variational minimization results in the original PDE to be solved replaced by a system of lineal equations $\mathbf{Ax} = \mathbf{b}$.

(3) Although the matrix $\mathbf{A}$ is not sparse, it is possible to implement some stationary iterative methods for solving $\mathbf{Ax} = \mathbf{b}$ (such as Gauss-Seidel or SOR) in a fast and efficient way. This is done recursively, by using the ideas behind the FFT, and results in algorithms consuming $O(N \log N)$ memory and executing one iteration in $O(N \log^2 N)$ time, N being the number of plane waves used.

(4) These stationary iterative methods can be accelerated via Krylov subspace methods (such as conjugate gradient or GMRES) or multigrid methods.

(5) The best results are obtained by the GMRES method using a multigrid preconditioner with Gauss-Seidel relaxation steps.

[1] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos, Rev. Mod. Phys. **64**, 1045 (1992).
[2] F. Gygi, Europhys. Lett. **19**, 617 (1992).
[3] F. Gygi, Phys. Rev. B **48**, 11692 (1993).
[4] F. Gygi, Phys. Rev. B **51**, 11190 (1995).
[5] F. Gygi and G. Galli, Phys. Rev. B **52**, R2229(R) (1995).
[6] D. R. Hamann, Phys. Rev. B **51**, 7337 (1995).
[7] D. R. Hamann, Phys. Rev. B **51**, 9508 (1995).
[8] D. R. Hamann, Phys. Rev. B **54**, 1568 (1996).
[9] D. R. Hamann, Phys. Rev. B **63**, 075107 (2001).
[10] G. Zumbach, N. A. Modine, and E. Kaxiras, Solid State Commun. **99**, 57 (1996).
[11] N. A. Modine, G. Zumbach, and E. Kaxiras, Phys. Rev. B **55**, 10289 (1997).
[12] A. Devenyi, K. Cho, T. A. Arias, and J. D. Joannopoulos, Phys. Rev. B **49**, 13373 (1994).
[13] E. Fattal, R. Baer, and R. Kosloff, Phys. Rev. E **53**, 1217 (1996).
[14] G. J. Pearce, T. D. Hedley, and D. M. Bird, Phys. Rev. B **71**, 195108 (2005).
[15] J. M. Pérez-Jordá, Phys. Rev. A **52**, 2778 (1995).
[16] J. M. Pérez-Jordá, Phys. Rev. B **58**, 1230 (1998).
[17] J. M. Pérez-Jordá, J. Chem. Phys. **132**, 024110 (2010).
[18] J. M. Pérez-Jordá, J. Chem. Phys. **135**, 204104 (2011).
[19] J. I. Rodríguez, D. C. Thompson, P. W. Ayers, and A. M. Köster, J. Chem. Phys. **128**, 224103 (2008).
[20] D. Dundas, J. Chem. Phys. **136**, 194303 (2012).
[21] J. M. Pérez-Jordá, Phys. Rev. E **90**, 053307 (2014).

[22] K. Ito, ed., *Encyclopedic Dictionary of Mathematics* (MIT Press, Cambridge, Massachusetts, 2000), 2nd ed.
[23] Y. Saad, *Iterative Methods for Sparse Linear Systems* (SIAM Press, Philadelphia, 2003), 2nd ed.
[24] S. J. Farlow, *Partial Differential Equations for Scientists and Engineers* (Dover, New York, 1993).
[25] J. W. Cooley and J. W. Tukey, Math. Comput. **19**, 297 (1965).
[26] L. C. Evans, *Partial Differential Equations* (American Mathematical Society, Providence, 2010), 2nd ed.
[27] R. N. Bracewell, *The Hartley Transform* (Clarendon Press, Oxford, 1986).
[28] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in FORTRAN* (Cambridge University Press, Cambridge, 1992), 2nd ed.
[29] R. S. Varga, *Matrix Iterative Analysis* (Springer, Berlin, 2000), 2nd ed.
[30] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial* (SIAM, Philadelphia, 2000), 2nd ed.
[31] F. Jensen, Theor. Chem. Acc. **104**, 484 (2000).