UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

# Performance improvement of eigenvalue computations in first-principles methods in physics

Master's Thesis

Master's Degree in Cloud and High-Performance Computing

AUTHOR: Mellado Pinto, Blanca

Tutor: Román Moltó, José Enrique

ACADEMIC YEAR: 2022/2023

## Thanks

Thanks to all the teachers who, during my Bachelor's and Master's studies, showed their dedication to the sharing of their knowledge and were an inspiration to me.

Among them, special thanks to my tutor, José E. Román Moltó, who was always available for any doubt or problem I had while writing this thesis.

Thanks to Alejandro Molina Sanchez, who provided the test cases and kindly answered our questions about Yambo via email.

Finally, I would like to thank all my loved ones, especially my parents, my sister, Łukasz and Clara. Thank you for always believing in me.

Special thanks to Jandro and Vanesa for answering my questions about physics and for their friendship over the years. Thanks to Celia for the advice and the time we spent together in the lab.

## Abstract

The work focuses on the Yambo software, an open source program that implements first-principles Green's function-based methods to describe the properties of excited states in realistic materials. Among these methods is the Bethe-Salpeter equation, which is the main focus of this work. In this equation appear the eigenvalues (energies) and eigenvectors (eigenstates) of a given Hamiltonian matrix. In Yambo, this eigenvalue problem can be solved by different methods. In one of them the complete diagonalization of the matrix is performed, either sequentially (LAPACK) or in parallel (ScaLAPACK). Another option is to calculate a small percentage of the eigenvalues using SLEPc. In this case, one can either create the Hamiltonian matrix explicitly, which in this application is dense, or alternatively operate with an implicit matrix. Our work is aimed at improving the performance, both computational and memory usage, of the different methods mentioned, exploiting parallelism at the level of both distributed memory (MPI) and graphics processors (GPUs).



**Keywords** Parallel Computing; Eigenvalue Computation; SLEPc; Yambo; Bethe-Salpeter.



## Resum

El treball se centra en el programari Yambo, un programa de codi obert que implementa mètodes de primers principis basats en la funció de Green per a descriure les propietats d'estats excitats en materials realistes. Entre aquests mètodes està l'equació Bethe-Salpeter, que és el focus principal d'aquest treball. En aquesta equació apareixen els autovalors (energies) i autovectors (autoestats) d'una determinada matriu Hamiltoniana. En Yambo, aquest problema d'autovalors es pot resoldre amb diferents mètodes. En un d'ells es realitza la diagonalització completa de la matriu, en seqüencial (LAPACK) o en paral·lel (ScaLAPACK). Una altra opció és calcular un percentatge xicotet dels autovalors mitjançant SLEPc. En aquest cas, es pot crear la matriu Hamiltoniana explícitament, que en aquesta aplicació és densa, o alternativament operar amb una matriu implícita. El nostre treball està encaminat a millorar el rendiment, tant computacional com d'ús de memòria, dels diferents mètodes esmentats, explotant el paral·lelisme tant a nivell de memòria distribuïda (MPI) com de processadors gràfics (GPU).



**Paraules clau** Computació paral·lela; Càlcul d'autovalors; SLEPc; Yambo; Bethe-Salpeter.



## Resumen

El trabajo se centra en el software Yambo, un programa de código abierto que implementa métodos de primeros principios basados en la función de Green para describir las propiedades de estados excitados en materiales realistas. Entre estos está la ecuación Bethe-Salpeter, que es el foco principal de este trabajo. En esta ecuación aparecen los autovalores (energías) y autovectores (autoestados) de una determinada matriz Hamiltoniana. En Yambo, este problema de autovalores se puede resolver con diferentes métodos. En uno de ellos se realiza la diagonalización completa de la matriz, en secuencial (LAPACK) o en paralelo (ScaLAPACK). Otra opción es calcular un porcentaje pequeño de los autovalores mediante SLEPc. En este caso, se puede crear la matriz Hamiltoniana explícitamente, que en esta aplicación es densa, o alternativamente operar con una matriz implícita. Nuestro trabajo está encaminado a mejorar el rendimiento, tanto computacional como de uso de memoria, de los diferentes métodos mencionados, explotando el paralelismo tanto a nivel de memoria distribuida (MPI) como de procesadores gráficos (GPU).



**Palabras clave** Computación paralela; Cálculo de autovalores; SLEPc; Yambo; Bethe-Salpeter.

# Contents

# List of Figures

# Chapter 1

# Introduction

The main objective of this Master's thesis is to optimize the computation of the eigenvalues and eigenvectors for the Bethe-Salpeter equation in the software Yambo.

Yambo is a code developed as part of the MaX European Centre of Excellence (MAterials design at the eXascale). MaX is a joint effort of several leading European partners in materials science, physics and High Performance Computing (HPC) to advance the research into new materials, in preparation for a near future where exascale supercomputing will be available. In particular, Yambo is a first-principles code that predicts fundamental properties such as band gaps in semiconductors or optical properties.

The Bethe-Salpeter equation plays a role in Yambo in the prediction of the optical properties. In Yambo this equation is implemented in the matrix form, and requires to solve an eigenvalue problem. Yambo implements several solvers to obtain the eigenvalues, with varying levels of parallelism. One of them uses the SLEPc library to compute a few eigenvalues. Another one uses routines from the LAPACK and ScaLAPACK libraries to compute the full spectrum via diagonalization. None of the implementations of these solvers currently takes advantage of the GPU parallelism, which is otherwise available in other steps in Yambo.

We worked with two test cases, provided to us by Alejandro Molina Sánchez. One of them uses an approximation that simplifies the equation and its matrix. The other one is an example of the general case and generates a matrix with four times the elements of the first one. The methodology used was to analyze the latest available version of the code at the time this master thesis was presented (September 2023), and improve the performance using these test cases as a benchmark. In the process, we used HPC methods and linear algebra libraries, for both distributed memory and GPU parallelism.

The proposed problem is interesting as the matrices are complex, dense and use single precision arithmetic. The problems that Yambo solves also generate quite large matrices, and some machines run out of memory before the computation is finished. The matrices associated with the test cases also have interesting characteristics. The first one is Hermitian. The second one presents an internal structure that can be exploited to reduce the memory required for storage.

With these ideas in mind, the following chapters will sequentially describe the context of this work, the analysis of the current implementation, the proposed implementations, the results obtained, and finally some conclusions.

# Chapter 2

# Context

## 2.1 Yambo

Yambo defines itself as a "plane-waves first-principles code for calculating excited-state properties – such as quasiparticle energies and optical spectra – of solid-state systems within the framework of many-body perturbation theory (MBPT) and time-dependent density functional theory (TDDFT)" [1].

The methods discussed in this project lie within the scope of the simulation of the optical properties of materials, like the optical absorption spectra. Concretely, they have to do with the solution of the Bethe-Salpeter equation.

The optical absorption spectra is a measure of how materials react to the incidence of photons of different energies. It is related to the macroscopic dielectric function $\varepsilon_M$. This function can be obtained with different approaches. One of them involves solving the Bethe-Salpeter equation. This approach yields good results, especially in semiconductors and insulators [2].

To illustrate these concepts, we will use as an example the optical spectra of $SiO_2$, also known as Silica. Figure 2.1 shows a typical graph plotting the energy of the photons against the absorption. The red dots correspond to the experimental results. The dotted line shows the simulated results using an RPA (Random Phase Approximation). The black line shows also simulated results, but with the Bethe-Salpeter approach.

Knowing the optical properties of materials is greatly relevant in fields like material science, electronics or even biotechnology. Yambo is widely used by scientists to study the structure and behaviour of materials, as it provides a tool for running computer simulations that complement the results of practical experiments.

One of the applications of Yambo is to research new materials with interesting properties for the development of electronic components like sensors, diodes or transistors.

For example, in [5] the authors study how applying processes like surface modification via hydrogenation or fluorination, can change the optical properties of monolayer GaN, shifting its spectra to the ultra-violet region, as shown in Figure 2.2. The results open the possibility to develop optoelectronic devices based on GaN which operate in those light frequencies. The simulations were run using Yambo with the GW-BSE approach.

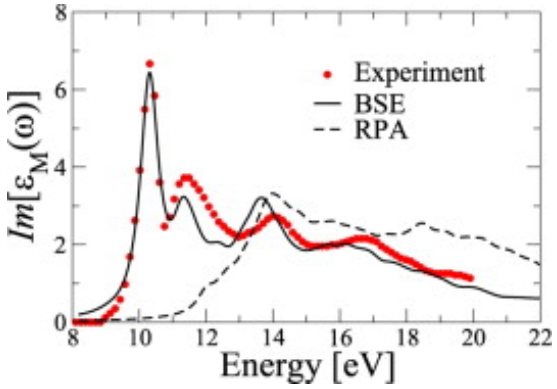


Figure 2.1: Absorption spectra for bulk silicon, taken from [1].This comparison plot was made by the authors using the experimental results of [3] and the calculations of [4].

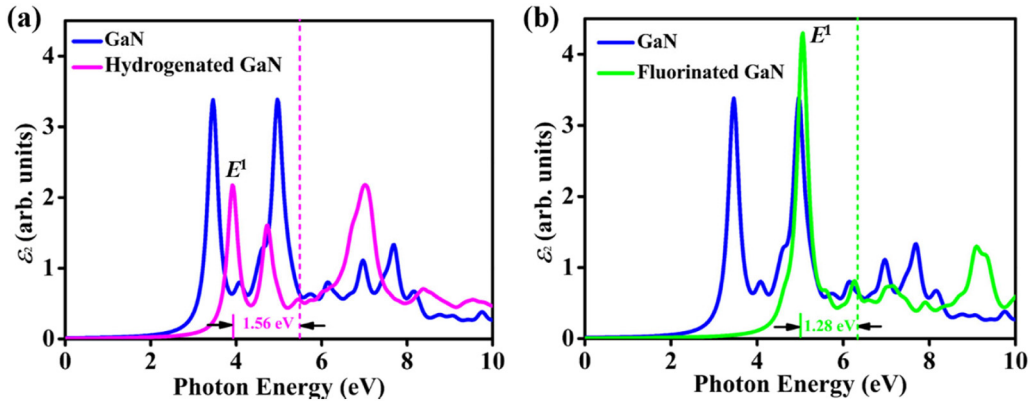


Figure 2.2: Shift of the optical spectra of monolayer GaN due to surface modification. Taken from [5].

Another example is [6], a study on the optical properties of single layer hexagonal boron nitride (hBN), an ultrathin two-dimensional material. Like graphene, it is a very promising material for the development of next-generation optoelectronic devices.

In practice, Yambo is a set of tools centered around the main executable Yambo, a complex software that can prepare and run different simulations given the properties of the material (which can be a molecule, surface, nanostructure, etc) and the methods and parameters desired for the task. These parameters must be tuned progressively, till achieving convergence, for accurate and significant results. Yambo also comes with utilities for performing conversion of formats from other DFT software packages for the input, like p2y (Quantum Espresso to Yambo) or a2y (Abinit to Yambo). Additionally it includes tools like ypp for postprocessing and visualization of the results.

The source code is mainly written in Fortran and C, while some tools like Yambopy [7], for the automation of convergence tests, are Python scripts. For this project, we tested some modifications in some of the Fortran functions inside the `bse` and `linear-algebra` modules related to the computation of the eigenvalues for the Bethe-Salpeter equation.

### 2.1.1 The Bethe-Salpeter equation

As mentioned above, one method to evaluate the optical properties in Yambo is to solve the Bethe-Salpeter equation [1].

The Bethe-Salpeter equation describes the bound state of two particles, in this case, an electron and a hole, bounded in a state called an exciton.

In some cases, photons can excite electrons and make them jump to the conduction band. When this happens, they leave a hole behind. The interaction between the electron and the hole creates a new "quasi-particle", the electron-hole. The electron and the hole, affected by their attraction and the external forces of the surrounding electrons, can bound as an exciton. Excitons have an impact on the optical properties of the material [8].

The Bethe-Salpeter equation in Yambo is solved in the matrix form by constructing a Hamiltonian-like matrix $H$ and solving an eigenvalue problem.

In section 2.2 we will discuss this problem and two methods which are currently used in Yambo to solve it.

The original publication of the equation by Salpeter and Bethe [9] already hints that the extension of the equation to more complex conditions should be straightforward, but prohibiting in terms of computational cost.

Indeed, the dimension of the problem (and its computational cost) grows with the size and complexity of the system described. In addition, as we mentioned, sometimes it is necessary to run a series of simulations to achieve valid results. If the size of the problem is large, it can take hours or even days to get such results.

Therefore, it is important to optimize and implement parallel versions of the solvers for this equation, in order to reduce the time needed to run the simulations, and to take full advantage of the resources of the machines where the code is run.

### 2.1.2 Optimization and parallel support

Yambo uses several libraries that provide optimized and/or parallel methods for linear algebra and distributed files such as:

- LAPACK: A standard library for linear algebra. It requires a BLAS implementation.
- BLAS: Basic linear algebra subprograms, optimized for each architecture.
- ScaLAPACK: Built on top of LAPACK and BLAS, it provides a similar interface and methods, but for distributed memory.
- FFTW: Optimized library for computing the discrete Fourier transform.
- NetCDF and HDF: Multidimensional scientific data formats, with support for parallel I/O.
- PETSc: Scalable parallel toolkit for scientific computation.
- SLEPc: Scalable parallel eigensolvers for large scale problems.

Yambo can also be configured to work with OpenMP [10], MPI [11] and CUDA [12]. There is currently an effort to extend the GPU support [13]. It is also possible to use Yambo in large clusters and supercomputers, as Yambo has been tested and optimized for HPC architectures.

A further analysis of the role of these tools in the currently implemented Bethe-Salpeter solvers can be found in chapter 3. More detailed descriptions of some of these libraries will be given in section 2.3.

## 2.2 Linear algebra problems

### 2.2.1 The linear eigenvalue problem

The eigenvalue problem seeks the computation of the eigenvalues, a set of scalars associated to a system of equations, commonly written in matrix form. We will focus on the linear eigenvalue problem, in which the equations of the system are all linear. This problem frequently appears in many areas of science and engineering, for example in applications where it is desired to find the conditions for the stability of a system.

The linear eigenvalue problem [14] is formulated as

$$Ax = \lambda x, \qquad x \neq 0, \tag{2.1}$$

where $A$ is the matrix, $\lambda$ represents the eigenvalues, and $x$ the eigenvectors. The eigenvalue problem can also be formulated as the generalized eigenvalue problem

$$Ax = \lambda Bx, \qquad x \neq 0. \tag{2.2}$$

#### Left eigenvectors

In equations (2.1) and (2.2) $x$ appears on the right and is therefore called the right eigenvector. Sometimes left eigenvectors are considered as well. Then

$$y^T A = \lambda y^T, \qquad y \neq 0 \tag{2.3}$$

or, in the complex case:

$$y^* A = \lambda y^*, \qquad y \neq 0. \tag{2.4}$$

Computing the left eigenvectors of a matrix $A$ is the same problem as computing the right eigenvectors of matrix $A^T$. If the matrix is symmetric,

$$A = A^T, \tag{2.5}$$

and the left and the right eigenvectors are the same.

Analogously, if the matrix is complex, computing the left eigenvectors of $A$ is the same problem as computing the right eigenvectors of matrix $A^*$. If the matrix is Hermitian,

$$A = A^*, \tag{2.6}$$

and the left and right eigenvectors are the same.

Here the superscript $^T$ denotes the transpose of a matrix or a vector, and the superscript $^*$ denotes the conjugate transpose.

#### Properties of matrices and eigenvalues

The eigenvalues of a diagonal or triangular matrix are the values on the diagonal. The eigenvalues of a matrix can be obtained by reducing the matrix to this forms by applying transformations.

Equation (2.7) shows an example of the eigenvalues of a diagonal matrix.

$$\begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{2.7}$$

Equation (2.8) shows an example of the eigenvalues of an upper triangular matrix.

$$\begin{bmatrix} \lambda_1 & a_{12} & a_{13} \\ 0 & \lambda_2 & a_{23} \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{2.8}$$

Other matrices are useful as intermediate forms. Matrices are commonly transformed to Hessenberg or tridiagonal forms as a step towards the computation of the eigenvalues.

A matrix is a Hessenberg matrix if all elements below the first sub-diagonal or above the first super-diagonal are zeros. Equation (2.9) shows an example of an upper Hessenberg matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \tag{2.9}$$

A matrix is tridiagonal if all its elements except the diagonal, the first sub-diagonal and the first super-diagonal are zeros. Equation (2.10) shows an example of a tridiagonal matrix.

$$\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \tag{2.10}$$

Transformations of matrices to simpler forms are usually made using orthogonal and unitary matrices, because that guarantees the numerical stability of the resulting algorithms.

A matrix is orthogonal if, when multiplied by its transpose on any side, it equals the identity matrix.

$$AA^T = A^T A = I \tag{2.11}$$

Analogously, for complex matrices, a matrix is unitary if it proves that multiplication by its Hermitian transpose equals the identity matrix.

$$AA^* = A^* A = I \tag{2.12}$$

Not all matrices can be diagonalized. An example class of diagonalizable matrices are normal matrices. A normal matrix satisfies

$$A^* A = AA^*. \tag{2.13}$$

**Uniqueness and realness**

Eigenvalues are not necessarily unique. Eigenvalues can be computed as the roots of the characteristic polynomial

$$\det(A - \lambda I) = 0, \tag{2.14}$$

which can be obtained transforming the initial equation of the eigenvalue problem (2.1) to

$$(A - \lambda I)x = 0. \tag{2.15}$$

If the matrix $A - \lambda I$ is singular, it will have a non-zero solution $x$, and the eigenvalues of $A$ are the roots of the polynomial. The characteristic polynomial has degree $n$, where $n$ is the dimension of matrix $A$. Therefore, matrix $A$ will have $n$ eigenvalues, but not all of them need to be unique.

Algebraic multiplicity expresses how many times a root is repeated in the characteristic polynomial, in other words, how many "copies" of the eigenvalue will appear when the eigenvalues are computed. If the algebraic multiplicity of an eigenvalue is one, it is called a simple eigenvalue. In physics literature, multiple eigenvalues are usually called degenerate.

Geometric multiplicity expresses the number of linearly independent vectors corresponding to an eigenvalue. Geometric multiplicity can be more than one if the eigenvalue is not simple; if it is simple, only one eigenvector corresponds to that eigenvalue. An eigenvalue with corresponding linearly dependent vectors is called defective. A matrix is also called defective if it has linearly dependent eigenvectors.

If all the eigenvectors $x_1, \ldots, x_n$ are linearly independent, and $X = [x_1, \ldots, x_n]$, $X$ is non-singular and can be inverted. Then if $D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$,

$$AX = XD \tag{2.16}$$

can be transformed into

$$X^{-1}AX = D. \tag{2.17}$$

Consequently, if all the eigenvectors are linearly independent, the matrix is diagonalizable.

A matrix with all distinct eigenvalues is diagonalizable, as all the eigenvectors must necessarily be linearly independent [14, p.5].

Because the roots of polynomials can be real or complex, eigenvalues can also be real or complex, even if the matrix is real.

Real symmetric and Hermitian matrices have real eigenvalues only [14, p.25].

### 2.2.2 Direct methods for the eigenvalue problem: full diagonalization

Direct methods seek to obtain the eigenvalues via factorization and transformations of the matrix. These methods give exact solutions in a finite number of steps. To be more precise, on computers, they give answers with an error inherent to computer calculations and float point arithmetic, caused among other factors by rounding errors.

Diagonalization methods for obtaining the eigenvalues of a matrix usually follow a 2-step scheme [15].

The algorithm starts with a matrix A, which is real symmetric or complex Hermitian. First, the matrix is reduced to tridiagonal form. Then, the eigenvalues are computed by further reducing the matrix to diagonal form.

#### Reduction to tridiagonal form

The matrix A is reduced to real tridiagonal form T using a unitary similarity transformation.

If the matrix is real symmetric, the transformation has the form

$$A = QTQ^T. \tag{2.18}$$

If the matrix is Hermitian, it has the form

$$A = QTQ^*. \tag{2.19}$$

In both cases $T$ is real.

**Computing the eigenvalues**

Next, the eigenvalues are computed from the tridiagonal matrix by reducing it to diagonal form, with transformation

$$T = PDP^T, \tag{2.20}$$

where $D$ is a diagonal matrix storing the eigenvalues, and $P$ is orthogonal and contains in its columns the eigenvectors of $T$. The eigenvectors of the original matrix A are the columns of the matrix $QP$, so a final matrix-matrix product is required to obtain the eigenvectors.

The factorization can be done in different ways, depending if we are interested only in the eigenvalues or in both the eigenvalues and the eigenvectors. Some available methods are:

- The QR algorithm, based on performing QR factorizations.
- Standard inverse iteration, or its successor, MRRR [16] (multiple relatively robust representations)
- Divide and Conquer: The Divide and Conquer algorithm by Cuppen [17] uses recursion. It is commonly used in parallel implementations [18]. When the recursion process reaches small-sized matrices, it may use the QR algorithm.

**Non-Hermitian matrices**

The method described above has a limitation. It does not work with a non-Hermitian matrix, as it is not always possible to diagonalize a non-Hermitian matrix. However, based on the property that all matrices can be transformed to triangular form [19, p. 171], a method exists to compute the eigenvalues of a non-Hermitian matrix via a transformation of the matrix into the Schur form [15]. The Schur form is a triangular matrix in the complex realm; or the real Schur form, a quasi-triangular matrix with 1x1 or 2x2 blocks on the diagonal.

**Computational cost**

The diagonalization method allows to compute all the eigenvalues of a matrix. However, as the process describes, a reduction to tridiagonal form (with cubic cost) and a subsequent transformation to diagonal (the cost is variable) is required, as well as a matrix-matrix product (again with cubic cost). Therefore, this method can be computationally expensive and slow. When the size of the problem increases, the cost of the operations involved grows accordingly. For large matrices, it can be prohibitive.

### 2.2.3 Iterative methods for the eigenvalue problem: the Krylov-Schur method

Iterative methods start with an initial solution, and make successive approximations, till convergence is achieved (convergence is not always assured to happen).

The Krylov-Schur method [20][21] is an iterative method which combines the projection of the matrix onto a Krylov subspace, with the Krylov-Schur restart.

### Projection methods

Projection methods compute a partial number of the eigenvalues of a matrix, by projecting the matrix onto a smaller subspace $\mathcal{V}$ and solving the eigenvalue problem in it.

$$H = V^T A V \tag{2.21}$$

- $A$ is the original matrix, of dimension $n$.
- $H$ is a smaller matrix, of dimension $m$.
- $V$ is a $n \times m$ matrix whose columns $v_i$ constitute an orthogonal basis of subspace $\mathcal{V}$.

First the matrix $H$ is computed, and then the eigenvalue problem

$$H z_i = \theta_i z_i \tag{2.22}$$

is solved.

Some of the computed eigenvalues $\theta_i$ can be good approximations of the eigenvalues of the initial matrix. The eigenvectors $x_i$ of the original matrix can be approximated as $V z_i$.

### Krylov subspaces

Certain subspaces are better than others for projecting the matrix. The Krylov subspaces constitute a good basis for effective algorithms.

$$H = K^T A K \tag{2.23}$$

$K$ is a basis that generates the Krylov subspace $\mathcal{K}$. They are generated from an initial vector $x_1$ in the following way:

$$\mathcal{K}_m(A, x_1) \equiv span\{x_1, Ax_1, A^2x_1, ..., A^{m-1}x_1\} \tag{2.24}$$

where $K = [x_1, Ax_1, A^2x_1, ..., A^{m-1}x_1]$.

The basis $K$ for the Krylov subspace is not used in practical applications because of its poor conditioning. Instead, a basis $V$ with orthogonal columns is computed using the Arnoldi or Lanczos algorithm (for the symmetric case) to generate the same subspace.

**Algorithm 1:** Arnoldi

```
Data: A, x_1
Result: V_m
v_1 = normalize(x_1);
for j = 1, 2, ..., m do
    w = A v_j;
    w = orthogonalize(w, V_j);
    v_{j+1} = normalize(w);
end
```

Although omitted in this simplified version of the Arnoldi algorithm, the projected matrix $H$ is computed at the same time than $V$.

In an iterative algorithm, in each iteration a new vector of the subspace is computed, as well as a new column of both $H$ and $V$.

**Krylov-Schur restart**

As explained above, the matrices grow with each iteration, requiring more memory and computing time. If many iterations are required, the advantages of using a projection algorithm are negated. This indicates that the vector $x_1$ that generates the subspace $\mathcal{K}$ is not rich in the direction of the eigenvalues we are looking for. One solution is to restart the process after reaching the maximum number of iterations $m$, with a new vector $x_1$, for example computed from the approximate eigenvectors. This is called an explicit restart.

Instead of explicitly computing a new vector $x_1$, we can perform an implicit restart with better numerical stability. It also preserves the information about the wanted approximate eigenvectors contained in $\mathcal{K}$. The implicit restart performs an Arnoldi decomposition of order $m$ and then compacts it to $p$, $p < m$ using the implicitly shifted QR algorithm. The process of expanding and contracting the decomposition is repeated until the desired eigenvalues converge.

The Krylov-Schur restart achieves the same effect, but is more simple to implement. Instead of starting with an Arnoldi decomposition,

$$AV_m = V_m H_m + \beta v_{m+1} e_m^*, \tag{2.25}$$

where the last term carries information about how close $\mathcal{K}$ is to an invariant subspace, we perform its generalization, the Krylov decomposition

$$AV_m = V_m B_m + v_{m+1} b_{m+1}^*. \tag{2.26}$$

The Krylov decomposition can be also written in matrix form as

$$AV_m = \begin{bmatrix} V_m & v_{m+1} \end{bmatrix} \begin{bmatrix} B_m \\ b_{m+1}^* \end{bmatrix}. \tag{2.27}$$

As in the implicit restart, in a second step the dimension of the decomposition is reduced to $p$, $p < m$. For the reduction, it is first transformed, using orthogonal transformations, into a Krylov-Schur decomposition, a Krylov decomposition where $B_m$ is in the real Schur form. Then there is a reordering and a truncation to $p$. The process of expanding and contracting is repeated until all desired eigenvalues converge or a maximum number of iterations is reached.

After all the desired eigenvalues have converged, it is possible that more than these have converged. For example, we may want to compute ten eigenvalues, but in the course of the computation we obtain twelve converged eigenvalues. A tolerance can also be set for deciding when the eigenvalues have converged. Usually a tolerance of the order of $10^{-8}$ is sufficient.

**Computational cost**

The matrix-vector product $Av_j$ is the most computationally expensive part of the Arnoldi algorithm if the matrix is dense. The cost of this operation increases quadratically with the dimension of the matrix. Therefore, optimizing this single operation for the specific architecture could also improve the performance of the algorithm.

Another way to reduce the cost is to reduce the number of iterations by finding the ideal balance between the $m$ and $p$ parameters, which depends on each problem and is not known a priori.

## 2.3 Numerical software

### 2.3.1 SLEPc

SLEPc (the Scalable Library for Eigenvalue Problem Computations) [22] implements solvers for several kinds of eigenvalue problems. It is optimized for large and sparse problems in parallel architectures. SLEPc parallelism is based on the distributed memory programming model and MPI message exchange. Additionally it also provides GPU parallelism for some of the implemented methods.

The purpose of the library is to hide from the programmer the complexity of working with parallel structures and managing the communications between the processes.

SLEPc works with both real and complex arithmetic. It can also be configured to work in single or double precision. It is written in C, and has interfaces for C++ and Fortran.

SLEPc is built on top of PETSc [23][24][25] (Portable, Extensible Toolkit for Scientific Computation), and it follows the same programming paradigm and standards.

Both PETSc and SLEPc follow an object-oriented approach. Data structures like matrices and vectors are encapsulated in objects, which hide its internal structure, and define what operations can be applied to them. Solvers and auxiliary classes are also objects.

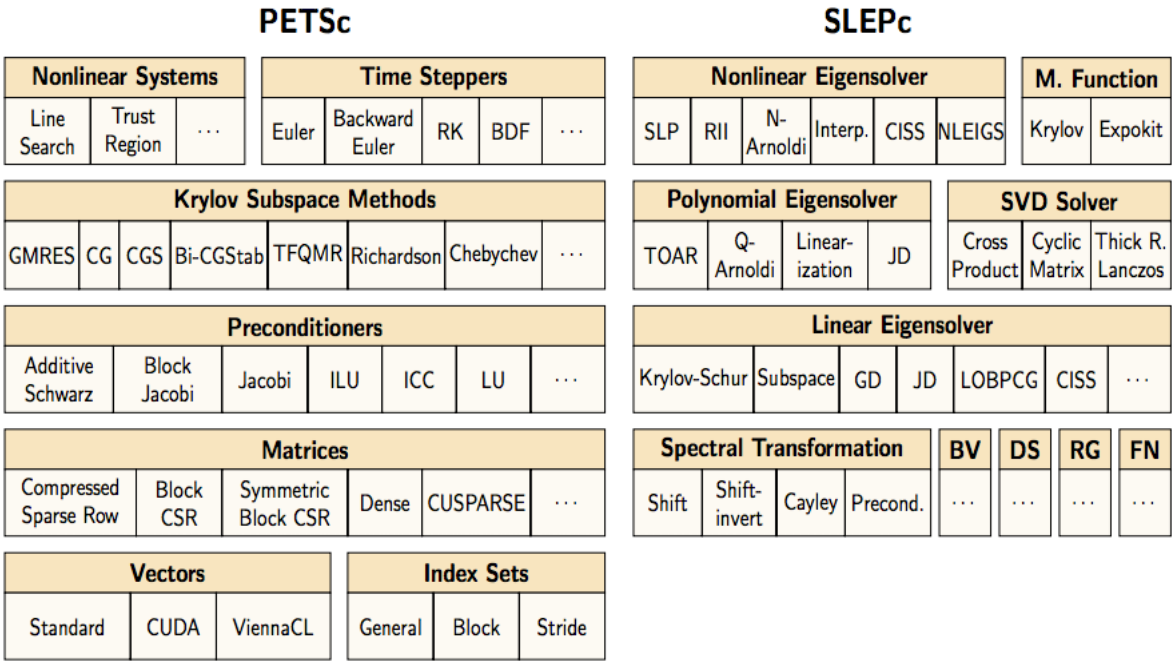


Figure 2.3: Solvers and methods available in PETSc and SLEPc, taken from the SLEPc user manual.

Some other practical aspects about the relationship between SLEPc and PETSc are:

- SLEPc installation requires a previous installation of PETSc in the system.
- Initializing SLEPc initializes also PETSc.

- The creation of matrix and vector objects is done through PETSc methods, as it uses the same structures and schema for parallelization. The matrices are distributed by rows among the processes, as in Figure 2.4.
- SLEPc internally uses PETSc's linear solvers and methods.
- SLEPc follows PETSc's conventions for logging and debugging, using `PetscViewer` objects and command line options to change the properties of objects and solvers at run-time .

SLEPc has modules for solving both the linear eigenvalue problem (EPS), and non-linear eigenvalue problems (NEP), from which the polynomial eigenvalue problem (PEP) is a particular case. SLEPc can also solve the singular value decomposition (SVD).

Most of SLEPc solvers are iterative solvers that use projections of the matrix onto certain subspaces. These are specially useful for large and sparse problems, or to extract only a few eigenvalues, instead of the whole spectrum, significantly reducing the computing time.

SLEPc can also apply spectral transformations, preconditioners, and extractions. These are aimed to improve the conditioning of the problem or to transform the spectrum, for example for computing eigenvalues from different regions, like internal eigenvalues.

An overview of the solvers and methods available in PETSc and SLEPc can be seen in Figure 2.3. SLEPc also interfaces to other libraries, like ARPACK, ScaLAPACK, ELPA or BLOPEX.

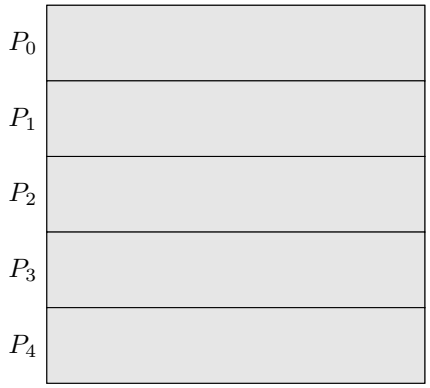


Figure 2.4: PETSc distribution of the matrix among the processes.

### 2.3.2 LAPACK

LAPACK [15] (Linear Algebra PACKage) is the successor of two linear algebra libraries: LINPACK [26], for solving systems of linear equations, and EISPACK [27], for solving eigenvalue problems. These libraries implement column-oriented algorithms to preserve the storage and memory addressing format of FORTRAN, making them more efficient. However, these libraries were not efficient enough for parallel computers using shared-memory.

LAPACK optimized linear algebra operations for shared-memory by using the block version of the same algorithms. This approach takes into account hierarchical memory, avoids moving data and makes addressing more efficient.

The block operations need to be very efficient, and it can only be achieved if they are optimized for the specific architecture where they will be run (cache levels, memory access, etc). The solution was to make

LAPACK a high level library whose routines call a BLAS code (Basic Linear Algebra Subprograms) for the basic block matrix operations, like the matrix-vector product. The BLAS code is specific and optimized for the architecture of the machine. This approach was also used by its predecessors, LINPACK and EISPACK.

BLAS is in nature a set of specifications for providing vector and matrix operations and it is organized in three levels:

- Level 1 BLAS: Scalar, vector and vector-vector operations (used in LINPACK and EISPACK)
- Level 2 BLAS: Matrix-vector operations
- Level 3 BLAS: Matrix-matrix operations (used in LAPACK)

The use of level 3 BLAS in LAPACK is one of the keys for its efficiency in shared-memory architectures.

LAPACK provides routines for matrix operations, matrix factorization, linear algebra solvers, least squares, eigenvalue solvers and many others.

The convention for naming the routines also comes from LINPACK and EISPACK. The names can be divided in three parts.

- The first letter indicates the type of arithmetic: S for single precision, D for double precision, C for complex numbers in single precision and Z for complex numbers in double precision.
- The next two letters indicate the type of matrix: HE for Hermitian matrices, TR for triangular matrices, etc.
- The following letters describe the operation: MM for the matrix-matrix product, MV for the matrix-vector product, etc.

For example, DGEMM is the routine for matrix-matrix multiplication of two general matrices in real double precision arithmetic.

The Hermitian diagonalization is called xHEEV in LAPACK, where x its replaced by the type of arithmetic. For complex arithmetic and single precision, the routine is CHEEV. Internally, the tridiagonalization step is performed by the routine CHETRD (for dense Hermitian matrices).

Computing the eigenvalues from the tridiagonal matrix can be done in several ways. If the eigenvectors are not computed, it uses an algorithm called square-root free QR (routine CSTERF). If the eigenvectors are computed, it uses the QR algorithm (routine CSTEQR). If instead of calling CHEEV, we call CHEEVD, the eigenvalues will be computed using Cuppen's divide and conquer algorithm.

### LAPACK and BLAS implementations

LAPACK is provided by the universities of Tennessee, California Berkeley and Colorado Denver. Currently it has a modified BSD licence, and it is sponsored by MathWorks and Intel. LAPACK is also included in commercial software, as it is allowed by its licence.

The BLAS specification has different implementations. OpenBLAS is a open-sourced, BSD licensed implementation with optimizations for different processors.

Intel implements its own BLAS in the Intel MKL (Math Kernel Library). MKL also implements LAPACK routines.

NVIDIA also has its own version of BLAS for GPU, cuBLAS. The analogous for LAPACK is cuSOLVER, which provides matrix factorization, dense and sparse linear algebra solvers and an eigenvalue solver. Both are included in the NVIDIA Math Libraries.

LAPACK has also influenced many linear algebra libraries that came after it. Two other libraries based on LAPACK are MAGMA (subsection 2.3.5), for GPU, and ScaLAPACK (subsection 2.3.3), for distributed memory.

### 2.3.3 ScaLAPACK

ScaLAPACK [28] (Scalable LAPACK), is the distributed memory version of LAPACK, created by the same universities than built LAPACK. There is a publicly available distribution on netlib[i], and vendor-specific versions which are optimized for a specific architecture.

The interface of ScaLAPACK imitates LAPACK's, making it easy for the user to port a code with LAPACK routines to ScaLAPACK.

Internally, it uses PBLAS (a parallel, distributed memory version of BLAS), LAPACK and BLACS (Basic Linear Algebra Communication Subprograms). BLACS is a linear algebra oriented interface for message passing, and its responsible for the communications of data between processes.

ScaLAPACK uses a two-dimensional block cyclic distribution of the dense matrices among the different processes, as shown in Figure 2.5. In the image, process with id 0 would store blocks marked with "0". If the matrix is banded, a block distribution is used instead.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 |

Figure 2.5: ScaLAPACK block-cyclic distribution of the matrix among the processes.

The analogous to LAPACK complex Hermitian diagonalization routines are PCHEEV, and PCHEEVD to compute the eigenvalues using the divide and conquer algorithm.

### 2.3.4 ELPA

ELPA [29] (Eigenvalue soLver for Petaflop Applications) is a library designed for solving eigenvalue problems in petaflop HPC systems. It is developed by the Max Planck Society, the university of Wuppertal, and the Technical University of Munich.

[i] `www.netlib.org`

Because it is designed with that purpose in mind, it scales very well for large clusters and supercomputers. It relies on the ScaLAPACK library, and uses the same block distribution, but replaces its internal parallel routines with its own.

ELPA optimizes the most time-critical part of the diagonalization, the reduction of the matrix to tridiagonal form. It offers two methods, one which performs it in one step, and a two-step method, which has shown very good results in scalability and performance. Diagonalization from the tridiagonal matrix is implemented in ELPA with the divide and conquer algorithm [30].

ELPA can solve the symmetric and Hermitian eigenvalue problem with real and complex matrices. It also supports all major architectures, including GPU.

### 2.3.5 MAGMA

MAGMA [31] (Matrix Algebra on GPU and Multi-core Architectures) is a collection of libraries developed by the University of Tennessee, in collaboration with universities of Berkeley and Colorado-Denver. It is also supported by Intel, AMD, NVIDIA, and MathWorks.

The objective of this libraries is to offer LAPACK based routines for heterogeneous systems with multicore CPUs and GPUs. The interface is similar to the one in LAPACK, but internally it implements hybrid algorithms that use both CPU and GPU parallelism.

MAGMA splits algorithms in a set of tasks and schedules them on the most appropriate available hardware components. For example, it will assign critical non parallel tasks to CPU and level 3 BLAS tasks to GPU. MAGMA's GPU parallelism gives especially good results for dense linear algebra.

As part of the MAGMA collection we can also find MAGMA SPARSE, with solvers that support sparse matrices.

MAGMA BATCH is a library for factorization (LU, QR and Cholesky) of many small matrices in parallel.

There are two alternative methods for diagonalization in MAGMA, one implementing the QR algorithm (`magma_cheevdx`) and one implementing divide and conquer (`magma_cheevd`).

Routines also have three versions, one where the matrix is initially stored in the CPU memory, one where it is stored in GPU and one for multiple GPUs. Most routines have also an interface for Fortran.

# Chapter 3

# Initial analysis

The initial approach to the project was to study the characteristics of the problem and the strategies and performance of the existing code. The goal was to understand what functions have the most impact on the performance in order to later optimize them.

This analysis was made based on two test cases that were provided to us. The first one solves the Bethe-Salpeter equation with all terms. The second one makes an approximation. As a result, in each of them the matrix has different size and characteristics.

## 3.1 Test cases

The test cases are simulations of the optical properties of $BiI_3$ that generate large matrices. A test case consists of a folder with all the databases[i] and files required for the test, and the scripts to run it.

The tests are described in a specification file, with a Yambo specific format, in which all the parameters for the test are defined. From now on, we will refer to this file as the Yambo specification file. The Yambo specification file also controls what methods will be run. For example, in our case `optics`, and `bss` (Bethe-Salpeter Solver), among others.

A Yambo run consists of a series of steps. The results from each step are stored in databases, and used in the next step of the calculations. If a parameter is changed and it affects the calculations for a step, that step and all the following ones must be rerun. However, databases from non-affected previous steps will be reused, saving some time.

Although we are interested in running only the Bethe-Salpeter Solver, first we must run one time all the previous steps. This process takes quite a long time[ii], but all subsequent runs are much faster.

### 3.1.1 Generation of the test

The process for generating the test (Figure 3.1), involves previously running an external DFT code to generate the input databases for Yambo. A variety of options of DFT codes are available, for example, Quantum Espresso or the software Abinit. Since sometimes a conversion of the format of the data is

[i] A database in Yambo is a set of files in the non-readable for humans netCDF format

[ii] Generating the the largest test took two full days in Tirant.

needed, Yambo provides tools like p2y, or a2y. In our case, the input was generated with Quantum Espresso and converted with p2y. This generates a `SAVE` folder.

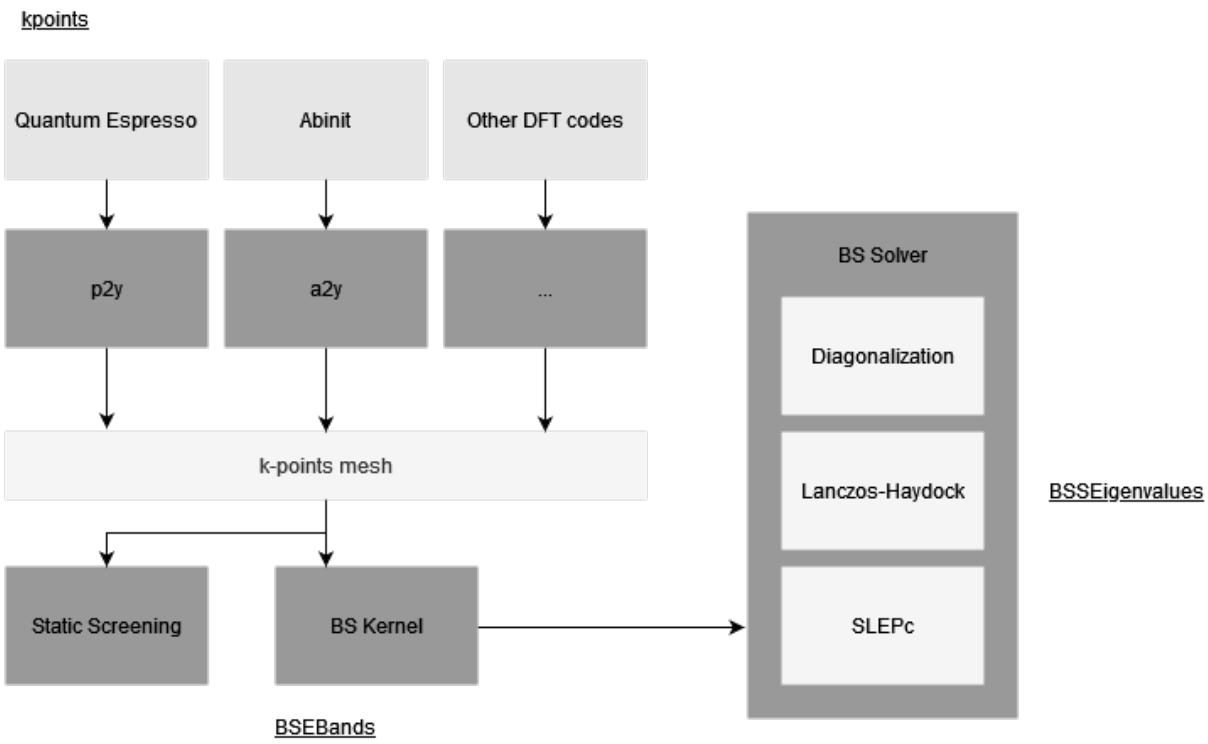


Figure 3.1: Generation of the test.

Once the input is prepared, we must run Yambo without options on the parent directory to the `SAVE` folder to prepare the databases for a later use. This set-up step runs a series of sampling and correction techniques.

At this point, we are prepared to run Yambo with the Yambo specification file. In our case, this concludes with creating the Bethe-Salpeter kernel and running the Bethe-Salpeter solver. Creating the kernel is the slowest part of the process, but once it is created, we can run the Bethe-Salpeter solver without doing it again.

Yambo also allows to use a folder, referred to as the job folder, to store intermediate computations and databases. Different folders can be used to keep the computations for different simulations.

### 3.1.2 Modifying the size of the test

As part of the process, we also generated the tests in smaller sizes, to be able to quickly test the correctness of the implementations made.

To change the sizes of the matrices in the test, we changed two parameters in the specification of the test. The first one was the size of the grid in the Quantum Espresso specification. We also changed

the parameter BSEBands in the Yambo specification file. Additionally, computing a larger number of eigenvalues does not change the size of the matrix, but it increases the time needed to run the test if we are using an iterative solver.

Changing the size of the grid requires to run all the computations again, including generating the input on Quantum Espresso. Changing the BSEBands only requires to run again Yambo, but the kernel is generated again.

### 3.1.3 Running the tests

Test cases are run by invoking the program in the command line with `yambo` and optionally specifying the following options:

- `-I`: Path to the folder containing the input databases with the characteristics of the material.
- `-F`: Path to the file containing the specifications for the run. This file can also be created invoking Yambo with specific options for it.
- `-C`: Path to the folder to save the output readable for humans.
- `-J`: Path to the job folder. A new job folder can also be created by giving it a name with this option.

Once the test is completed, we might want to examine the results or, in our case, we are interested in the correctness and performance of the solvers. In the output folder we will have the following useful files:

- Reports: Files starting with "r-". These files contain a report of all the steps in the Yambo run, including the time each step took to be completed. We are particularly interested in this information. Reports also show the parameters used in the Bethe-Salpeter solvers and the convergence results of the SLEPc solver.
- Logs: Files starting with "l-". We can keep track of the content of this file while Yambo is running to see the progress and issues. When we run Yambo in parallel with more than one process, we will have a `LOG` folder instead, with one file for each process.
- Output: Files starting with "o-". These files contain the results of the test, in a column format. The first column contains the energies of the photon. The rest of the columns contain the optical properties, and we can use them to plot the optical spectra. For example, we can plot the absorption spectra with the energies from the first column, and the second column, which represents the imaginary part of the macroscopic dielectric function. An example of this plot can be seen in Figure 3.2.

Among other methods, we used the plots made from the output to test the correctness of the modifications, and compare if the results are still the same after the modifications to the code. Although the eigenvalues can be slightly different when they are computed with different methods and software, if the difference between the eigenvalues is inside some tolerance, we consider the result to be acceptable.

### 3.1.4 Differences between the two test cases

The Bethe-Salpeter equation, in its general matrix form with all terms involved, has the following form for this problem [32]:

$$\begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{3.1}$$

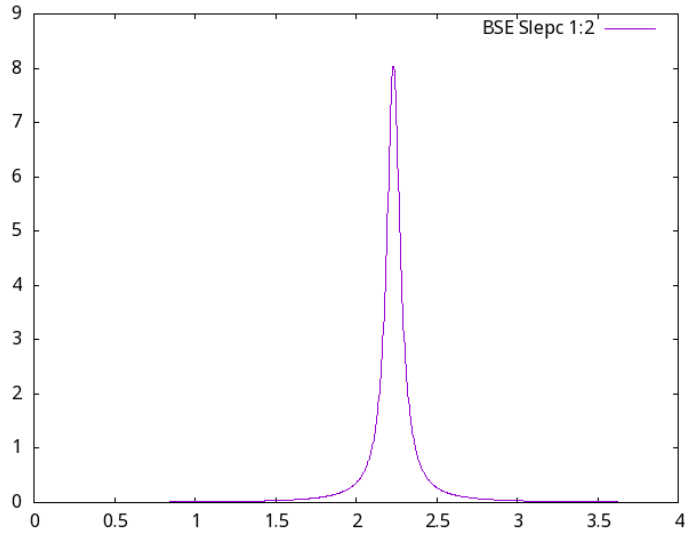


Figure 3.2: Absorption spectra from one of the test cases.

The block $R$ is a Hamiltonian corresponding to the resonant transitions, that happen from valence states to conduction states. The mirroring block $-R^T$ corresponds to the anti-resonant transitions, from conduction states to valence states.

Blocks $C$ and $-C^*$ are coupling terms between the transition blocks.

When the coupling terms are much smaller than the resonant and anti-resonant terms, they can be discarded, resulting in the matrix having only one block, R.

We worked with two cases, one discarding the coupling, and one keeping it.

#### Test case with Tam-Dancoff Aproximation

In this case, the Bethe-Salpeter equation is truncated and the coupling between the resonant and anti-resonant transitions is discarded. Only the electron transitions from the valence states to the conduction states are considered. This is called the Tam-Dancoff approximation (TDA) [33], and in some cases it does not describe the system accurately [34].

The evident result of using this approximation is that the dimensions of the matrix are smaller. The time needed to solve this test case is consequently greatly reduced.

To generate this case in Yambo we indicate `mode='resonant'` in the Yambo specification file.

The dimension of the matrix in the provided test is 30276, with a total of 916636176 elements, which accounts for 3.4GB of space in memory.

We will refer to this test case as the resonant case going forward.

#### Test case with coupling introduced

In this test the coupling terms are present. It gives a more accurate representation but it also generates a matrix with two times the dimension (four times the number of elements).

To generate a case with coupling we indicate `mode='coupling'` in the Yambo specification file.

The dimension of the matrix in this case is 60552, with a total of 36665447048943415 elements, and 13.7GB of space in memory.

We will refer to this test case as the coupling case.

## 3.2 Analysis of the matrices associated to the problem

### 3.2.1 Structure

The size of the matrix is not the only factor affected by the mode. Depending on the problem, two types of matrices appear, a Hermitian matrix in the resonant case, and a non-Hermitian matrix in the coupling case. Both matrices are square matrices.

The matrices are constructed in both modes by assembling a series of rectangular blocks, previously generated by Yambo in the step of computing the kernel for the solver. This kernel blocks are labeled by their mode, `'r'` for resonant blocks and `'c'` for coupling blocks. The code outside the Bethe-Salpeter solver is also parallelized by distributing these kernel blocks between the processes, and the number of them depends on the number of processes also. However, the parallelization inside the solvers does not follow the same approach. The parallelization strategies for the solvers will be explained in section 3.4.

#### Hermitian case

A Hermitian matrix appears in the resonant mode. The eigenvalues of a Hermitian matrix are real and the eigenvectors are orthogonal. The eigenvalues and eigenvectors can be solved through diagonalization.

The kernel blocks that form this matrix are all resonant, `'r'`. Because the matrix is Hermitian, it is necessary to compute (and store) only half of the elements when constructing it. When a kernel block is written to the matrix, its values are copied transposed also. In Figure 3.3 these kernel blocks are shown.

#### Non-Hermitian case

In the coupling mode the matrix is non-Hermitian. This property limits which algorithms can be used to solve the eigenvalue problem, as some methods like diagonalization require the matrix to be Hermitian.

The matrix is formed by four blocks[iii].

$$H = \begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix}. \tag{3.2}$$

To construct the matrix, two types of kernel blocks appear: resonant blocks `'r'` and coupling blocks `'c'`. Resonant kernel blocks are used to construct the elements of the matrix from the top left block and the bottom right block. Coupling kernel blocks are used to construct the top right block and the bottom left block.

The resulting matrix is four times the size of the Hermitian matrix in the resonant mode. It has the following characteristics:

[iii] Notice that this blocks are not the same as the previously referred to in this section as "kernel blocks". We have given that name to the structures used in Yambo for storing the computations made in the kernel step that are later written to the matrix, to avoid confusion. The "blocks" described here, and in the sequel, refer to the characteristic structure of the matrix for the Bethe-Salpeter equation.

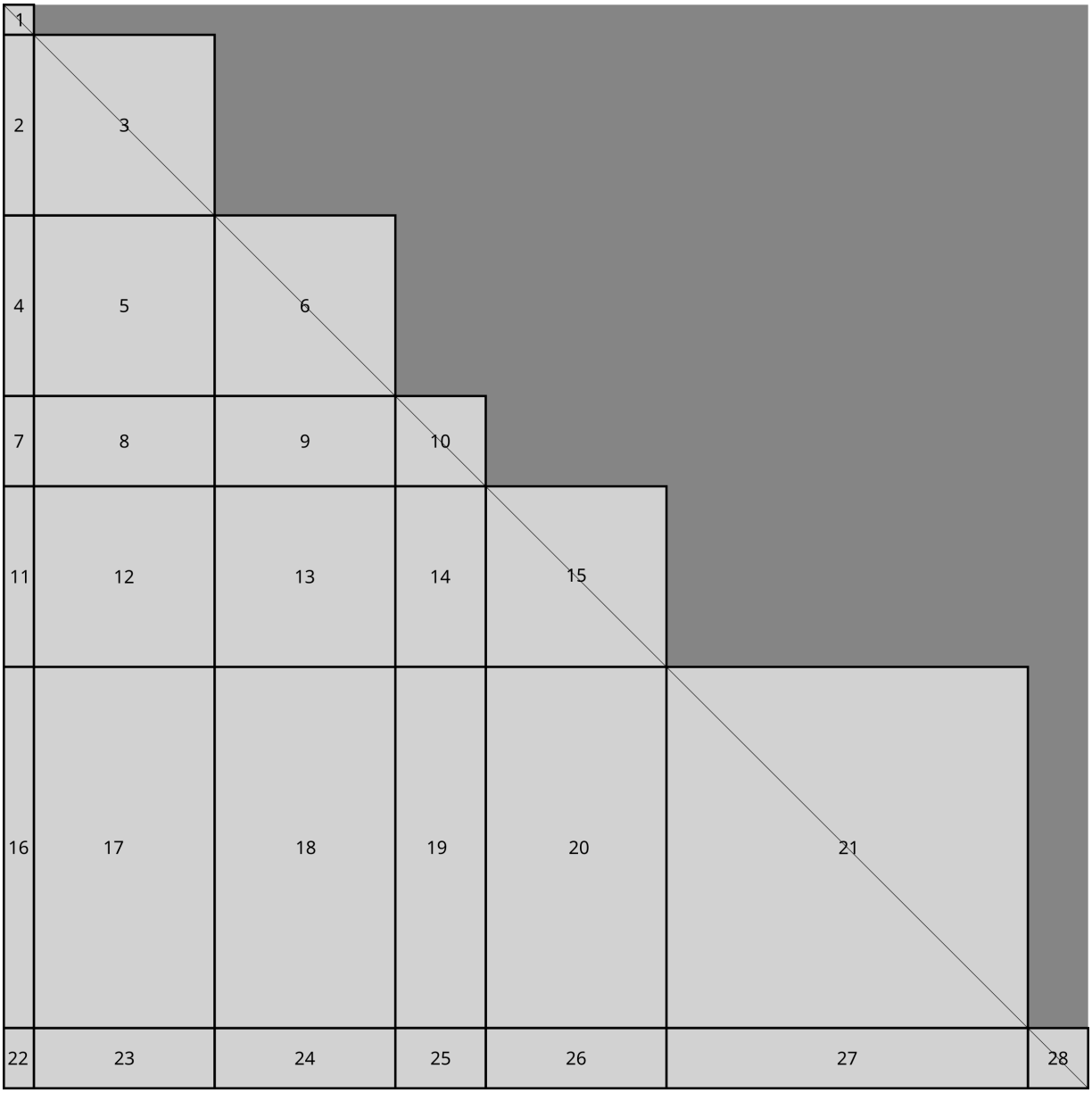


Figure 3.3: Positions of the kernel blocks with the elements of the Hermitian matrix.

- The top left block R is Hermitian.
- The top right block C is symmetric.
- The bottom left block is the Hermitian transpose of C scaled by -1.
- The bottom right block is R transposed and scaled by -1.

Consequently, although the non-Hermitian matrix is larger, it can be constructed from just a quarter of its elements.

### 3.2.2 Format

The matrix associated to the problem is in both cases a dense matrix. In practice, the memory requirements can be a limitation when the problem is of a large dimension.

The format of the matrix depends on the solver used.

The diagonalization solver constructs the full matrix from the kernel blocks and distributes a copy of it to all the processes. When the matrix is large and the processes are running on the same node, the memory of the node will not be enough to hold all of the copies. This has been confirmed in our tests, where trying to run the code with more processes in the same node resulted in Yambo being killed because of a lack of memory. A better approach would be to distribute the matrix between the processes.

The SLEPc solver, on the current implementation, offers to select between two types of matrix to prioritize speed for smaller problems and memory management for larger problems.

- Explicit matrix: Constructs the matrix explicitly from the kernel blocks, requiring more space in memory.
- Shell matrix: A special type of matrix available in PETSc. The matrix is not stored explicitly, instead, its operations are defined, which requires less space in memory. In this case, it is required to provide a function that performs the matrix-vector product to use it in the solver.

## 3.3 Current implementation

Yambo offers three methods for computing the eigenvalues:

- Diagonalization: The diagonalization solver computes all eigenvalues and eigenvectors of the matrix. This is the best approximation to obtain an accurate optical spectra. However, for large problems it requires significantly longer time than the other solvers. Internally, it uses the LAPACK library to compute the eigenvalues. This solver also has a parallel version in which LAPACK is replaced with ScaLAPACK.
- Lanczos-Haydock: Computes a subset of the eigenvalues till a threshold accuracy is reached. The limitation of Lanczos-Haydock is that it does not compute the eigenvectors. The optical absorption spectra can be computed with just the eigenvalues, but the eigenvectors are needed for later calculations, so it is interesting to compute them also.
- SLEPc: Computes a subset of the eigenvalues and their eigenvectors (for example the smallest ones). It is faster but the number of eigenvalues must be specified beforehand which results in a less accurate approximation if fewer than necessary are requested.

In the documentation, Yambo advices to use the Lanczos-Haydock method to obtain the eigenvalues and then use SLEPc to obtain the eigenvectors of the dominant features. A comparison of the results of both approaches is shown in Figure 3.4.

As we have mentioned it is possible to select some parameters for the solvers. The options are available through the Yambo specification file.

- `BSSmod`: Selects the solver among the different available options.

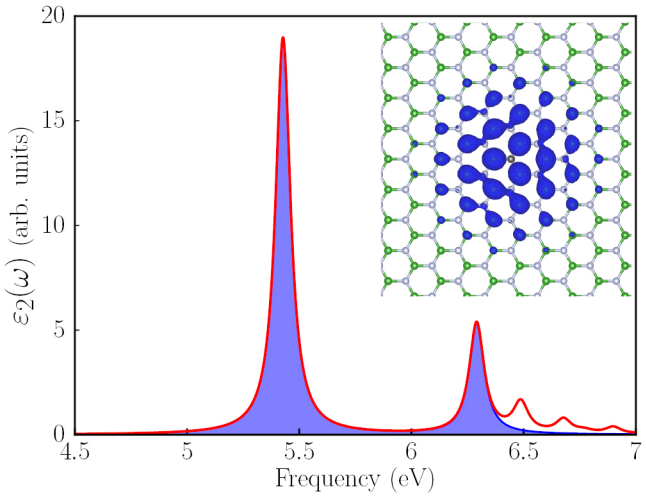


Figure 3.4: Comparison of SLEPc and Lanczos Haydock. The image, taken from [35] shows the optical absorption spectra of mono-layer hBN, obtained with Yambo. The blue shaded region shows the spectra obtained by computing only two eigenvalues with SLEPc. The red line corresponds to the results of the Lanczos-Haydock solver. In this case, two eigenvalues can describe the most prominent features.

  - 's': SLEPc
  - 'd': diagonalization
  - 'h': Lanczos-Haydock

The following options are specific to the SLEPc solver:

- `BSSNEig`: The number of eigenvalues to compute.
- `BSSEnTarget`: The energy target for the solver.
  - If there is no target and a preconditioner is used, it computes the eigenvalues closest to 0.
  - If there is no target and no preconditioner is used, it calculates the smallest eigenvalues.
  - If there is a target, the eigenvalues are ordered by distance to it.
- `BSSSlepcShell`: Use a shell matrix, this is the default option.
- `BSSSlepcMatrix`: Use a explicit matrix.
- `BSSSlepcNCV`: The dimension of the subspace.
- `BSSSlepcApproach`: The solver to be used inside SLEPc, there are three options available right now. SLEPc offers more solvers, but only these three are offered by Yambo for this problem. By default the Krylov-Schur solver is used.
  - 'Krylov-Schur'
  - 'Generalized-Davidson'
  - 'Jacob-Davidson'

- `BSSSlepcPrecondition`: The preconditioner to use. It must be selected among the following options. The default option is 'preonly+jacobi'
  - 'none'
  - 'preonly+jacobi'
  - 'bcgs+jacobi'
- `BSSSlepcExtraction`: If it is defined, it will aply an extraction. The available options are:
  - 'ritz'
  - 'harmonic'

## 3.4 Current parallelization strategy

Currently the code for the solvers is parallelized with a distributed memory strategy using MPI and the SLEPc and ScaLAPACK libraries.

### 3.4.1 Diagonalization

The serial version of the code uses the LAPACK library. In the parallel version the routines are exchanged with the corresponding ones in ScaLAPACK.

The code for the diagonalization of the matrix is only parallel for the Hermitian case as ScaLAPACK does not support yet the diagonalization of non-Hermitian matrices.

In the Hermitian case, the parallel version of the code is run if Yambo has been configured with ScaLAPACK and more than four CPUs are used. This apparently arbitrary number is due to the size of the smallest ScaLAPACK grid defined in Yambo, which is 2 rows by 2 columns.

Although Yambo has GPU support for other steps of the computations, there is currently no implementation using GPU parallelism for any of the solvers.

### 3.4.2 SLEPc solver

Currently, Yambo can be configured to link to an existing installation of SLEPc. SLEPc is used inside Yambo to compute a small range of eigenvalues of the Bethe-Salpeter equation, instead of the whole range.

In the case of the SLEPc solver, the distribution of the memory and the communications between the processes are handled by the library using MPI. The strategy for distributing the matrix and the vectors between the processes is implemented differently in Yambo than in SLEPc.

SLEPc and PETSc distribute by the matrix by rows. At the moment of the creation of a matrix, the number of local rows that each process will have can be either specified, or left for PETSc to decide by passing `PETSC_DECIDE` as the parameter. An MPI communicator can also be passed to the function to specify among which processes the matrix will be distributed.

In Yambo, before the matrix for SLEPc is constructed, its elements are distributed among the processes in the kernel blocks. The number and distribution of the blocks depends on the number of processes.

To convert between both distributions, there is a communication of the data structures to all the processes before and after calling the SLEPc solver. This process adds an overhead in time and memory.

The parallelization described above only refers to the case of using an explicit matrix. When a shell matrix is used, the matrix is not constructed explicitly in SLEPc.

Although SLEPc supports the use of GPU, currently Yambo does not offer this possibility to the users.

## 3.5 Conclusions of the analysis

From our analysis we noticed that there is no GPU implementation for any of the Bethe-Salpeter solvers in Yambo. SLEPc offers GPU support and we can use it to our advantage. We can also introduce GPU parallelism on the diagonalization solver.

Diagonalization is the slowest method for solving the Bethe-Salpeter equation in Yambo. We will try to improve the performance in distributed memory also, by using state-of-the-art libraries designed for large problems.

Lastly, we found that memory is a limiting factor. The structure of the non-Hermitian matrix allows us to use nested storage, and we will explore this approach to reduce the memory required to store the matrix in SLEPc.

# Chapter 4

# Implementations

In this chapter we will describe the methods used and the adaptations made to the code to optimize the performance of the Bethe-Salpeter solver. The changes were made inside a fork of the Yambo repository. We tested the changes on different machines by recompiling the Yambo source code with our modifications, and the required configuration. For version control, we used git, a tool that becomes fundamental when working with large codes, and aided us to maintain control of the different versions.

As part of the process we configured and compiled Yambo with libraries which are not supported in the current Yambo configuration. These implementations were tests and if they were to be implemented, they would require changes in both the configuration and the flow and dependencies of the code.

From our analysis in chapter 3, we needed to adapt to the following limitations, arising from the characteristics of the problem:

- Single precision: Although a double precision version is available, Yambo is mostly used in single precision. We made the choice to work on that version and limit the libraries and methods used to those who work in single precision, to avoid architectural issues or the need to convert between formats.
- Complex numbers: Similarly, we are limited to use libraries that solve the eigenvalue problem for complex matrices, as the matrix associated to the problem is complex.
- Dense matrix: The methods which apply to sparse matrices to reduce memory and optimize performance are not useful in this case.
- Large problems: A special care must be paid to the use of memory, as we have experienced it to be a limiting factor when running the simulations.

In the following sections we will describe the implementations that yielded significant results, with an initial remark on what changes to the code were needed to measure the performance.

## 4.1 Profiling

To measure the performance we used Yambo reports and SLEPc logs.

Yambo reports contained information about the time taken to run the solver. For the SLEPc solver, we separated this time in two, the time taken to create the matrix, and the time to run the solver. These two

processes are made by two different functions in the code. The goal was to find out whether creating the matrix to adapt to the specifications of SLEPc impacted the time significantly. In general, we found that, on large problems, the time to run the solver had a bigger impact, so we focused our efforts on improving that part of the code.

In SLEPc and PETSc there is a mechanism for allowing the user to set options at run-time via command options. These options can also be specified in the code.

Currently the code for the SLEPc solver in Yambo does not allow the user to input PETSc options. Instead the users must select the solvers, preconditoneres, etc from a limited set of options in the Yambo specification file. This approach is safer for the user and allows them to use the solvers without requiring any knowledge in the use of the SLEPc library. For the purpose of this project, we made changes to the code to be able to access the full possibilities of PETSc and SLEPc, extract logs and important information from SLEPc, and change the solver's characteristics in run-time, without changing the code.

Some of these options are evaluated at `PetscFinalize` or `SlepcFinalize`. Other options apply to the matrices or solvers and can even change the format of the data or the method used in the solver. To set up a solver with the command line options from the user we must call `EPSSetFromOptions` from the code. To set up the properties of the matrix with the user input we must call `MatSetFromOptions`.

The order in which we call `EPSSetFromOptions` is important. All options which are set in the code after the call will overwrite the ones from the user input.

We used the following PETSc options:

- `-log_view`: Prints a log with information about memory and time profiling, objects created, etc.
- `-log_view_gpu`: Prints additional information about GPU usage to the log.
- `-log_view_memory`: Prints additional information about memory allocation to the log.
- `-mat_view`: Prints information about a matrix.
- `-device_enable_cuda eager`: Initializes the GPU on `PetscInitialize` even if it is later not used. We used this option to get accurate times for GPU.
- `-mat_type`: Selects the matrix type, for example the sparse format. In this case the matrix is dense and it was set to `MPIDENSE`.

Additionally, we used the following SLEPc options:

- `-eps_view`: Prints information about the eigenvalue solver and the parameters used for the computation.
- `-eps_view_mat0`: Prints information about the matrix A in the eigenvalue problem. It can also write the matrix in a file.
- `-eps_type`: Selects the method for the linear eigenvalue solver.
- `-eps_error_relative`: Shows the computed eigenvalues and the error.

Additionally you can specify the format of the output, and the name of a file to save it by adding the optional parameters `<format>:<filename>:<options>:`.

In PETSc it is possible to tag a matrix with a command line option to later view its properties in run time. This is done in the code with function `MatViewFromOptions`.

We used this feature to extract the matrices and analyze them using Matlab, in order to check that the format of the matrix matched the expected structure and properties.

## 4.2 Optimizations to the Hermitian case

The focus on the Hermitian case was to improve the speed. We explored different solutions for distributed memory and GPU parallelism.

### 4.2.1 Diagonalization in CPU with ScaLAPACK and ELPA

The implementation of diagonalization in CPU currently uses LAPACK for the serial version. A parallel version is available only for the Hermitian case in ScaLAPACK.

Our first approach to improve the performance was to exchange the ScaLAPACK library with ELPA, as it has shown very good results and scalability for large matrices. An additional advantage of using ELPA would be the possibility to implement a parallel version also for the non-Hermitian case, as ELPA covers this case in its routines.

SLEPc offers an interface to both ScaLAPACK and ELPA. This allowed us to compare both libraries with no implementation required. Using the SLEPc interface instead of directly calling the functions in the libraries will add a small overhead, but the comparison will be fair as both of them will have the same overhead.

We used the command line option `-eps_type` to change the solver from the Krylov-Schur solver to the ScaLAPACK one with `-eps_type scalapack` or to the ELPA interface with `-eps_type elpa`.

### 4.2.2 Diagonalization in GPU with MAGMA

The diagonalization method in Yambo also doesn't have a GPU implementation yet. We modified the code for the serial version of the diagonalization using LAPACK, and ported it to MAGMA to run the diagonalization on the NVIDIA GPU. Additionally, we made a second version to run it on multiple GPUs, in our case, in two.

We used the interface of MAGMA for Fortran, so the functions start with `magmaf_` instead of `magma_` and all the parameters are passed by reference. We need to pass also an additional parameter for the error code, which is not needed in C, where this role is fulfilled by the return value of the function.

Firstly we initialize the library, with `magmaf_init`. This step also detects the GPUs available, and their characteristics, for example their architecture, capability, etc. When we are done using the library we must call `magmaf_finalize`.

Next, we substituted the LAPACK functions for the MAGMA ones. The interfaces are very similar, but some aspects change. The naming conventions are also maintained. We exchanged the LAPACK routine `CHEEVD` for the equivalent one in MAGMA, `magmaf_cheevd`.

The routine in MAGMA had the following parameters:

- `jobz`: A character indicating whether to compute also the eigenvectors.
- `uplo`: A character indicating which triangle of A to store, the upper or lower one.
- `n`: The order of the matrix.
- `A`: The matrix.

- `lda`: The leading dimension of the matrix.
- `w`: The vector where the eigenvalues will be written.
- `work`: The complex work space.
- `lwork`: The dimension of the complex workspace.
- `rwork`: The real work space.
- `lrwork`: The dimension of the real workspace.
- `iwork`: The integer workspace.
- `liwork`: The dimension of the integer workspace.
- `info`: Provides information about the result of the routine.

The interface of the function requires references to several workspaces and their dimensions. To know the dimension of the workspaces we called the function two times. In the first one, we passed -1 as an argument to the parameters for the dimension of the workspaces. Doing so prompts the routine to make a workspace query and return in the first element of the workspace, the required dimension of the workspace. The second time, we call the function passing the proper workspace dimension and the workspaces allocated in memory.

Finally, we made an implementation with MAGMA for multiple GPU devices, changing the routines to their multi-GPU version, `magmaf_chveed_m`. An additional parameter must be passed indicating the number of GPUs.

### 4.2.3 Krylov-Schur in GPU with SLEPc

The Krylov-Schur method is also not implemented in Yambo for GPU yet.

In SLEPc it is possible to exploit the GPU parallelism by changing the type of the matrix to `DENSECUDA`. This is one of the matrix types available in PETSc using NVIDIA GPU acceleration, along with `AIJCUSPARSE`, for sparse matrices. If the algorithm has an implementation for the matrix type, it will run in GPU.

While this change can also be made in the code, for our purposes, we modified the matrix type in run-time indicating the option `-mat_type densecuda` when Yambo was launched.

The GPU methods in PETSc follow a lazy mirror model. Data structures are copied from main memory to GPU memory. If an operation is available on the GPU and the data structures were previously modified in main memory, the data structures in GPU memory are updated and then the operation is performed. If the operation is to be performed on the CPU and the data structures in the GPU have been modified, the data structures in main memory are modified before the operation is performed. However, if a sequence of operations is performed only in the GPU (or CPU) one after the other, no updates are made between the memories [36].

## 4.3 Optimizations to the non-Hermitian case

The matrix is non-Hermitian when coupling is introduced. In this case, the matrix is also four times the size. Therefore, the memory is a limiting factor in this case. The time to compute the eigenvalues also increases significantly.

For the non-Hermitian case we focused on reducing the memory requirements for the SLEPc solver, taking into account the performance also.

The currently implemented solution to the memory limitations is to use a shell matrix to reduce the memory. A shell matrix is a PETSc object that represents the matrix, but does not store it explicitly. Instead, the programmer defines how to perform the algebraic operations which are required for the solver. For the Krylov-Schur solver, the shell matrix must provide a method to compute the matrix-vector product.

The downside for this implementation in Yambo is that it reduces the speed, because the access to memory in this operation without the explicit matrix is not optimal.

Our solution uses nested storage, taking advantage of the internal structure of the matrix described in section 3.2.1 to store only part of the matrix. In [37] structure-preserving algorithms for this particular matrix are discussed. We will adapt this approach to our case and to use PETSc's nest matrix object.

The goal is to reduce the memory required for the storage of the matrix, without excessively harming the performance of the solver. We will also test if this approach can compare in performance to storing the whole matrix.

### 4.3.1 Nested storage

As we have previously determined, the matrix has the structure shown in equation 4.1. We used a nest matrix to store only R and C explicitly.

$$H = \begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix} \tag{4.1}$$

**Nest matrix**

A nest matrix is a special matrix type `MATNEST` available in PETSc to represent that a matrix is formed of several nested submatrices.

The object is created through the routine `MatCreateNest`, indicating the following parameters:

- The MPI communicator for the matrix.
- The number of submatrices in a row
- An index set with the positions of the submatrices in the rows. We indicated NULL to make them contiguous.
- The number of submatrices in a column
- An index set with the positions of the submatrices in the columns. We indicated NULL to make them contiguous.
- An array of matrix objects with all the submatrices that will compose the nest matrix, ordered by row and then column. The submatrices can be explicit or shell matrices. The array can also contain NULL elements, to represent empty submatrices.

We used a nest matrix were blocks $R$ and $C$ are stored explicitly and blocks $-C^*$ and $-R^T$ are shell submatrices that use the already stored explicit blocks in their operations.

### Shell submatrices

PETSc provides a special type of shell matrices, representing the transpose and Hermitian transpose of a matrix, through routines `MatCreateTranspose` and `MatCreateHermitianTranspose`. Although we considered these structures as an option, they could not be scaled, so they could not represent a negative matrix as needed. So instead, we created two shell submatrices ourselves.

To create them we used the PETSc routine `MatCreateShell`. Its interface requires to provide the dimensions of the matrix and the local dimensions, even when the matrix will not be stored.

The Krylov-Schur solver in SLEPc for non-Hermitian matrices uses the matrix-vector product, both of the matrix and its transpose. To perform this operation on the nest matrix, we must provide a routine to compute both products, of each of the shell submatrices. These operations will internally use the data from the blocks explicitly stored.

To assign the operations to the shell submatrices we call `MatShellSetOperation`, passing as parameters the matrix, the operation, and a reference to the implemented routine that computes it.

The interfaces of the routines are defined by the type of operation and they must follow their specification. The interface allows to pass a context object to the routine, which is commonly used to pass the data needed for the product. In Fortran we needed to use a module to store the explicit blocks.

### Matrix-vector product

The matrix-vector product, defined as

$$y = Hx, \tag{4.2}$$

can be expanded to

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \tag{4.3}$$

resulting in equations

$$y_1 = Rx_1 + Cx_2 \tag{4.4}$$

and

$$y_2 = -C^*x_1 - R^Tx_2. \tag{4.5}$$

Products $Rx_1$ and $Cx_2$ using the explicit submatrices will be handled by PETSc, and we must define how to compute the products $-C^*x_1$ and $-R^Tx_2$ for the shell submatrices. For that purpose, we define two functions with the interface required by the operation, whose parameters are a matrix, a vector and a parameter where the resulting vector will be written.

Product $-C^*x_1$ is performed in the first function by calling `MatMultHermitianTranspose` with explicit submatrix $C$ and then scaling the resulting vector by -1 with `VecScale`.

Product $-R^Tx_2$ is performed in the second function by calling `MatMultTranspose` with the explicit submatrix $R$ and then scaling the resulting vector by -1.

### Matrix-vector product of the transposed

The Krylov-Schur method in SLEPc requires also to define a function for the product of the transpose.

The transposed matrix is constructed as

$$H^T = \begin{bmatrix} R^T & (-C^*)^T \\ C^T & (-R^T)^T \end{bmatrix} = \begin{bmatrix} R^T & -\overline{C} \\ C^T & -R \end{bmatrix}, \tag{4.6}$$

where it is important to notice that the coupling blocks have been exchanged, in addition to being transposed. [i]

The matrix-vector product of the transposed matrix follows

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} R^T & -\overline{C} \\ C^T & -R \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.8}$$

and can be expanded to the equations

$$y_1 = R^T x_1 - \overline{C} x_2, \tag{4.9}$$

$$y_2 = C^T x_1 - R x_2. \tag{4.10}$$

From these equations, the products $R^T x_1$ and $C^T x_1$ involve the explicit submatrices and are performed by PETSc. We need to define the routines for the products $-\overline{C}x_2$ and $-Rx_2$.

The product of $-Rx_2$ is performed using PETSc's routine `MatMult` and then scaling the resulting vector to -1.

The product of $-\overline{C}x_2$ requires to conjugate a matrix.

The product of a conjugated matrix and a vector

$$f = -\overline{A}z \tag{4.11}$$

can be transformed to

$$\overline{f} = -A\overline{z}. \tag{4.12}$$

This way, we avoid the need to conjugate a matrix, and perform the more economical conjugation of two vectors.

We conjugate the vectors using the PETSc function `VecConjugate()`. The result is also scaled to -1. It is important that the vector $x$ remains unchanged outside of this routine, for the following products.

**Nest matrix assembly process**

The whole process of assembling the nest matrix can be summarized in the following steps:

1. Create the explicit submatrix and fill the values for block $R$.
2. Create the explicit submatrix and fill the values for block $C$.
3. Create the shell submatrix for block $-R^T$.
4. Define operations `MATOP_MULT` and `MATOP_MULT_TRANSPOSE` for the shell submatrix of block $-R^T$.

[i] Since C is symmetric and R is Hermitian, we could develop further this expression as

$$H^T = \begin{bmatrix} \overline{R} & -\overline{C} \\ C & -R \end{bmatrix}, \tag{4.7}$$

However the matrix-vector products of $\overline{R}$ and $C$ are handled by PETSc with the explicit matrices. A possibility would be to define shell matrices for all the blocks to optimize those products.

5. Create the shell submatrix for block $-C^*$.
6. Define operations `MATOP_MULT` and `MATOP_MULT_TRANSPOSE` for the shell submatrix of block $-C^*$.
7. Create an array containing the submatrices for blocks $R$, $C$, $-C^*$ and $-R^T$, in this order.
8. Create the nested matrix from the array.

# Chapter 5

# Results

For validation and performance testing of our solutions, we used the two test cases presented in section 3.1. The tests were run on two machines: tirant, and gpu.

## 5.1 Test environment

The characteristics of the two machines are listed in the Table 5.1. Some interesting aspects and motivations for using the selected machines are explained in the next subsections.

### 5.1.1 Tirant v3

Tirant v3 is a node of the Spanish supercomputing network (RES). Its current Linpack Rpeak performance is 111,8 Teraflops.

One advantage of using Tirant is the ability to use multiple nodes, each with a process. This was essential because running large Yambo tests in parallel, with multiple processes on the same node, resulted in the node's memory being strained and the system killing Yambo because it was unable to allocate the required memory. The cause of this issue is that Yambo uses replication of data structures across processes to avoid more complex communication between them during computation. Using a multi-node cluster allowed us to run the coupling case, which involves the largest matrices, in parallel.

In general, communications between processes in different nodes are slower than between processes in the same node. However, the inter-node communication speed depends on the type of network and its specifications. In this case, the Tirant cluster is mounted with Mellanox InfiniBand FDR10, a high performance network.

Tirant is a shared system which uses slurm as a queue manager[i]. The tests were submited as batch jobs to Tirant via scripts with `#SBATCH` directives, indicating the number of nodes, processes, etc to schedule for the job. To submit a job, it is also required to indicate a maximum time limit, that the user must decide according to the estimated time that it will take for the job to run.

[i] In tirant 50% of the compute hours are reserved for the RES. The rest is available for "local" Tirant users. Users must submit their jobs through slurm, although it is possible to run small tasks in the front-end nodes, as long as they take less than 10 minutes.

Projects have different limitations to the resources depending on their requirements. The limitations for this project were 72 hours for each batch job and 64 nodes.

Table 5.1: Characteristics of the machines

| | **tirant v3** | **gpu** |
|---|---|---|
| CPU model | Intel® Xeon® E5-2670 (Sandy Bridge) | Intel® Core™ i9-7960X |
| Number of nodes | 336 | 1 |
| Sockets per node | 2 | 1 |
| Physical cores per socket | 8 | 16 |
| Logical cores per socket | 16 | 32 |
| Physical cores per node | 16 | 16 |
| Logical cores per node | 32 | 32 |
| Memory per node | 32 GB DDR2 | 64 GB |
| Operating system | OpenSuSE Leap 42.3 | Ubuntu 18.04.6 LTS (Bionic Beaver) |
| Number of GPUs | - | 2 |
| GPU model | - | NVIDIA® Quadro RTX™ 5000 |
| GPU architecture | - | Turing |
| GPU compute capability | - | 7.5 |
| GPU cores | - | 3072 |
| GPU memory | - | 16 GB GDDR6 |
| GPU performance (S.P) | - | 11.2 TFLOPS |
| Network | Mellanox InfiniBand FDR10 | - |
| C compiler | ICC 19.0.4 (Intel) | NVC 23.5 (NVIDIA) |
| Fortran compiler | IFORT 19.0.4 (Intel) | NVFORTRAN 23.5 (NVIDIA) |
| MPI | Intel MPI Library 2019.4 | OpenMPI 3.1.5 |
| CUDA version | - | 11.4 |
| BLAS/LAPACK | MKL 2019.4 (Intel) | cuBLAS 11.4 (NVIDIA) |
| ScaLAPACK | 2.2 | 2.2 |
| SLEPc | 3.19 | 3.19 |
| ELPA | 2022.11 | - |
| MAGMA | - | 2.7.1 |
| Yambo | 5.1.0 | 5.1.0 |

We used `#SBATCH --exclusive` directive to reserve a full node, even when the number of processes required was less than the number of cores of the node. That way it is possible to ensure that no other jobs will be allocated in the nodes.

### 5.1.2 gpu

Gpu was used mainly to test the efficiency of the libraries and implementations for gpu and multi-GPU parallelism, as it has two NVIDIA Quadro RTX GPU cards. This machine also has more more memory per node than tirant, but the number of processes is limited by the number of cores of the node (16 physical and 32 logical). Additionally we also used gpu to run smaller test cases.

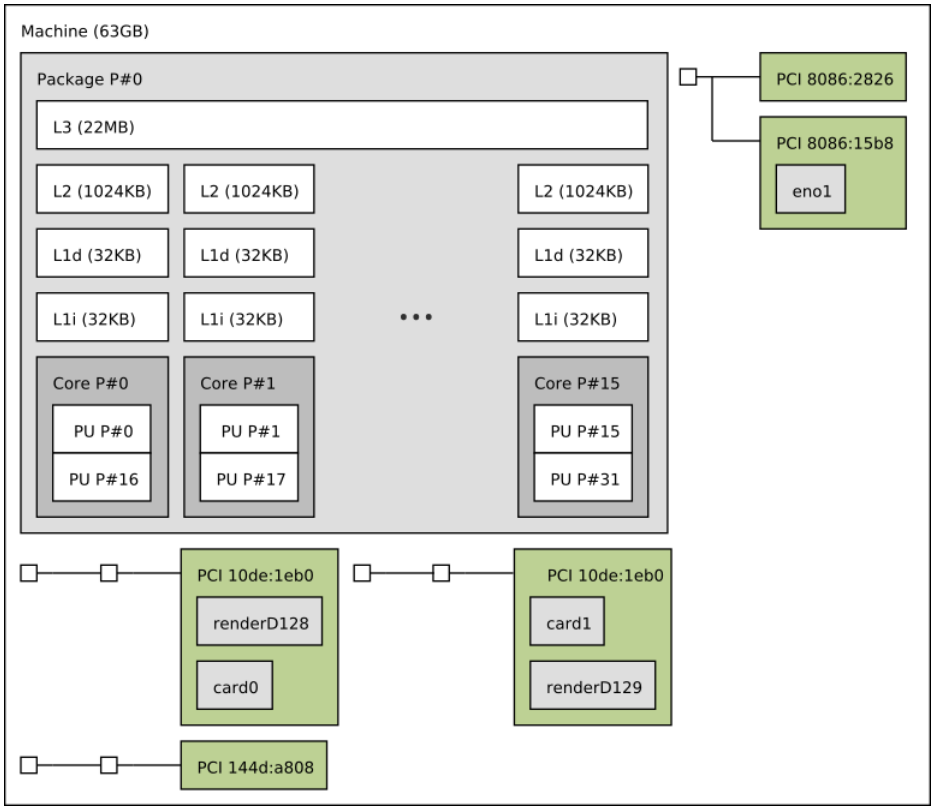


Figure 5.1: Topology of the machine gpu.

## 5.2 Hermitian case

In this section we will describe the results of the implementations on the Hermitian case, which were tested on the resonant test case.

### 5.2.1 Diagonalization in CPU with ScaLAPACK and ELPA

Figure 5.2 presents the results of the tests comparing the performance of the ELPA and ScaLAPACK libraries on the diagonalization solver, which computes all the eigenvalues of the matrix. The tests were run on tirant, using one process per node due to the memory limitations explained above.

Although ELPA was expected to have a better performance than its counterpart due to its optimization of the computation for clusters, in this case we observed no significant difference with the number of processes tested, which was at maximum 24 nodes from tirant. Further tests with more processes could show larger differences, as ELPA is developed for large clusters and supercomputing architectures. It is also suggested that a possible approach to increasing the performance of ELPA is to configure the library to use a more efficient CPU instruction set.

In addition, the diagonalization method was shown to scale well with the two libraries, with similar results for both.

An unexpected result was that the version with ScaLAPACK interfaced from SLEPc was faster than the

current implementation using ScaLAPACK directly, even though the SLEPc interface adds an overhead due to the extra layer. In some cases, the current implementation also ran out of memory when the interfaced one didn't. This indicates that the current implementation with ScaLAPACK can be further improved.

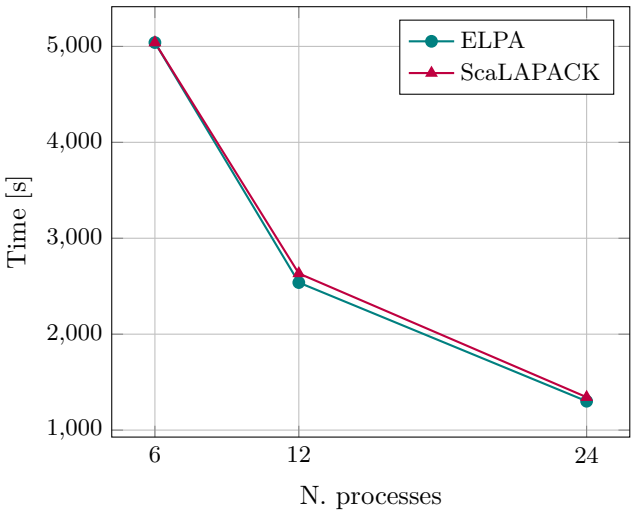


Figure 5.2: Results of performance using the ELPA and the ScaLAPACK diagonalization solvers, interfaced from SLEPc.

### 5.2.2 Diagonalization in GPU with MAGMA

An interesting result of our work was the parallelization in GPU of the diagonalization method in Yambo using the MAGMA library. In the current version of Yambo, the diagonalization method can only run sequentially or in parallel using distributed memory. In addition, the use of GPU showed good results on the test architecture, being able to reduce the time to compute all the eigenvalues from five hours and a half to four minutes. The results are presented in seconds in Table 5.2.

Even better results were obtained in a second version using the two available GPUs in the machine, though the difference in time in this size of the test is seconds. Nevertheless this result shows that it is possible to scale the computation using several GPUs. The communication hardware between the GPUs would also impact this time. We also observed that the time for the matrix creation remained the same, while the time reduction happened when computing the eigenvalues.

Although the results are promising, we also need to consider the memory limitations of the GPUs, because the matrix needs to be loaded in the memory of the GPUs. In this case, the memory available was enough for this size of the problem, but it may not be the case with a larger matrix.

Table 5.2: Performance comparison using LAPACK, and MAGMA with one and two GPU's

| | LAPACK (Sequential) | MAGMA 1 GPUs | MAGMA 2 GPUs |
|---|---|---|---|
| Matrix creation | 17.4s | 17.7s | 17.3s |
| Solver | 20340.0s | 235.6s | 178.4s |

### 5.2.3 Krylov-Schur in GPU with SLEPc

Figure 5.3 shows the results using the Krylov-Schur method on the GPU from SLEPc. As expected, the GPU versions with one and two processes are more efficient than their CPU counterparts, and the differences increase as the number of eigenvalues computed increases. This problem therefore has a very good GPU parallelization capability.

Although in figure 5.3 the difference is not visible, in figure 5.4 we can observe in detail how the solver performs when run on one, and two GPUs with distributed memory. The single GPU version is initially faster for 100 eigenvalues, due to the overhead of using two GPUs that need to communicate data between them. However, as more eigenvalues are computed and the computations become more extensive, using two GPUs proves to be faster.

As with the diagonalization solver, the use of the SLEPc solver on the GPU is limited by the memory characteristics of the GPUs used.

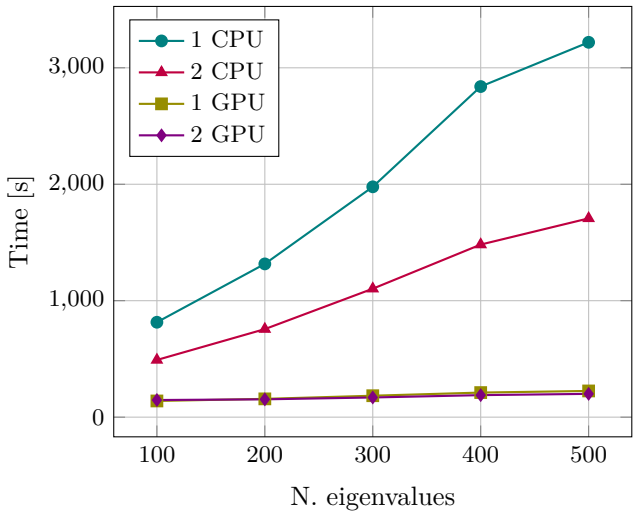


Figure 5.3: Comparison of time performance of the Krylov-Schur SLEPc solver with one and two processes on CPU and GPU.

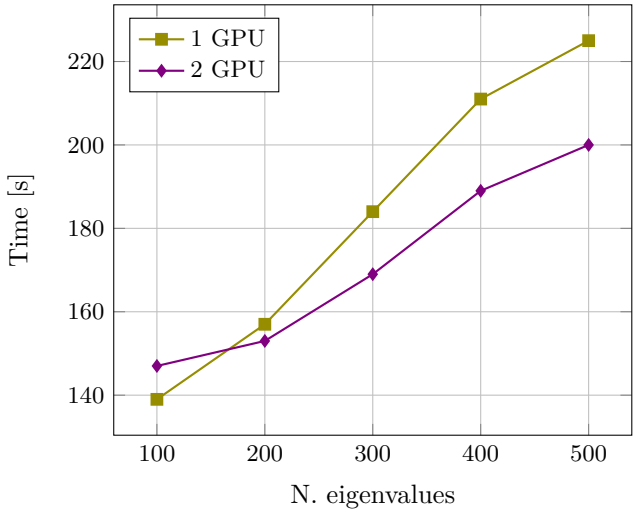


Figure 5.4: Comparison of time performance of the Krylov-Schur SLEPc solver with one and two GPUs

## 5.3 Non-Hermitian case

Next, we will describe the results of the implementation of the nested storage for the non-Hermitian case, which was tested on the coupling test case.

### 5.3.1 Nested storage

The coupling test case, which generates a considerably larger matrix and requires more available memory, could not be run when the matrix was explicitly stored. However, we were able to run the test case using the shell matrix and our implementation of the nest matrix.

Having demonstrated the usefulness of these two implementations in reducing the memory required to run large tests, we compared the time performance of the solver using the nest and shell matrix formats, which we a priori expected to be better for the former.

We ran a sequential test on Tirant, computing five eigenvalues. As expected, the nest matrix format is faster, taking five hours and eighteen minutes, while the shell matrix test took five hours and forty-four minutes to complete the solver computations. These results are presented in Table 5.3.

We then ran parallel tests on Tirant with a different number of processes. In these tests we noticed that with both matrix formats and only with some number of processes the solver reached the maximum number of iterations and some of the eigenvalues didn't converge. This behaviour would need to be investigated further to understand the cause.

In order to be able to compare the times with the explicit matrix, we reduced the size of the test and thus the matrix. In this case, the cases were run on the machine's GPU to avoid waiting for the assignment of nodes in Tirant. The number of eigenvalues calculated was also five. In figure 5.5 we can compare the three approaches. Using the shell matrix format is the slowest method, and the nest and explicit formats give comparable time results. Using more than fourteen processes caused the machine to run out of memory when the explicit matrix was used.

Table 5.3: Comparison of times for the Krylov-Schur SLEPc solver on Tirant with different types of matrix format.

| | Shell matrix | Nest matrix | Explicit matrix |
|---|---|---|---|
| Matrix creation | 0.840s | 286.28s | Out of memory |
| Solver | 20940s (5h 45m) | 19080s (5h 18m) | Out of memory |

In Figure 5.6 we can see the differences in the time taken to construct the matrix. The shell matrix takes close to zero time to construct. This is because it is only a virtual structure. The nest matrix is created in significantly less time than the explicit matrix because it only needs to write half the values into the matrix.

Running the same test in parallel with a larger number of eigenvalues (e.g. 50) resulted in the same behaviour as the first test in Tirant, where some of the eigenvalues didn't converge. This behaviour occurred with all matrix formats.

In summary, the nest matrix format could allow the user to significantly reduce the memory required for the tests, while retaining the speed benefits of explicit matrix storage. While the shell matrix format further reduces memory requirements, the nest matrix is faster and offers a different trade-off between memory constraints and speed.

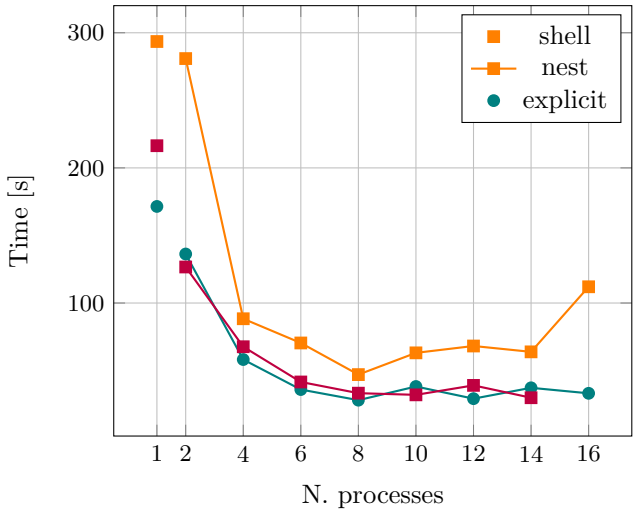


Figure 5.5: Time comparison of the Krylov-Schur SLEPc solver using the different storage formats

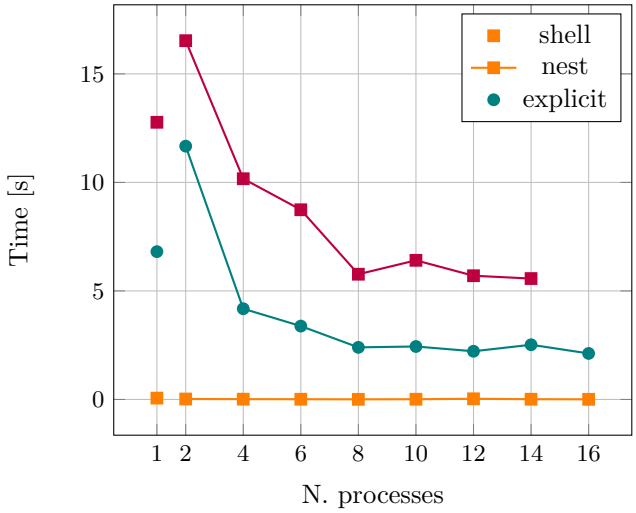


Figure 5.6: Comparison of the time for creating the matrix using the different formats on the Krylov-Schur SLEPc solver.

## 5.4 Further work

**Smoothing the differences in data distribution between SLEPc and yambo**

The distribution of data between processes is very different in Yambo and in SLEPc. Yambo uses a parallelization scheme to create the matrix, which distributes blocks of the matrix to the processes. In SLEPc, the matrix is distributed by rows. When Yambo calls the SLEPc solver, it must first create and distribute the matrix in the SLEPc scheme.

**Communication of data between processes**

Another factor that adds to the memory load is the copying of the entire data structures between all of the processes. This approach can speed up the computations, but it also makes the code less scalable. An example is the communication of the matrix to all of the processes. Limiting the communication of data between processes could reduce the memory allocation of the processes.

**BSEPack**

A package which would be worth exploring is BSEPACK [38], developed by Meiyue Shao and Chao Yang. It is a specific purpose solver targeting the Bethe-Salpeter eigenvalue problem, for distributed memory HPC systems.

**Refining the implementation of the nested storage**

As we described in subsection 4.3.1, the matrix-vector product with the transposed matrix can be refined taking into account the symmetry of the block $C$ and that the block $R$ is Hermitian. This is one idea that could be tested to see if it would improve the performance .

### Finer tuning of ELPA and ELPA with GPU

The results obtained from comparing the performance of the ELPA library with ScaLAPACK didn't match our theoretical expectations. In our tests, we used a shared cluster and limited out tests to work with 24 nodes maximum, due to the time limitations and the waiting time for nodes growing with the number of them requested. As it was mentioned, the results obtained using more processes could be different. ELPA also has support for GPU, which would be interesting to test and compare with the performance of MAGMA.

### Parallel implementation of the diagonalization for non-Hermitian matrices

Currently there is no parallel implementation of the diagonalization method for non-Hermitian matrices. In [39], a method is proposed to extend the ELPA diagonalization method for symmetric matrices to skew-symmetric matrices. The matrix in the Bethe-Salpeter equation (the general case) is related to a skew-symmetric matrix, and the motivation for their work. This method is available in the ELPA library and could be used to provide a distributed memory version of the diagonalization method to the Yambo users.

# Chapter 6

# Conclusions and discussion

In summary, in this Master's thesis we explored different optimizations of the solvers for the computation of eigenvalues on the Bethe-Salpeter equation in the software Yambo. In practice, this problem deals with large matrices that are dense, complex, and typically in single precision. The importance of optimizing the code comes from the need to provide faster results and to be reliable and efficient on any architecture, including clusters and supercomputers. This has an impact on the overall computational time in Yambo, which in turn could accelerate the research of new materials.

The tests we ran and the execution of the code in general are subject to the limitations imposed by the machine characteristics: bandwidth, memory, architecture... Among them, we found that memory was the most limiting factor.

We propose a solution to reduce the size of the matrix in memory in the non-hermitian case problem, taking advantage of the fact that the structure of the matrix allows us to use nested storage and physically store only half of the values. The results we obtained with this approach show that it can also be compared in terms of time with the use of the explicit matrix, thus there is no need to greatly compromise the time performance in order to save memory. In our analysis, we also observed the possibility of additional improvements related to memory management and cooperation between libraries.

Another result of this work was a first prototype and tests of GPU implementations for the eigenvalue computation, which are not yet available in Yambo. First, we implemented and tested the diagonalization in GPU using MAGMA. Second, we showed how to configure the SLEPc solver to run on GPU and compared its performance to the CPU option. In both cases we obtained very good results in terms of time improvement. We believe that it would be useful for users of Yambo to have this option to speed up the computations if they have a GPU available.

In addition, a comparison between ELPA's and ScaLAPACK's diagonalization performance did not show any major differences in the cases examined, although further tests with more processes or nodes may provide additional insight.

Finally, we proposed some additional ideas for improving the solvers, which would enable future work on this topic.

# Bibliography


[1] Andrea Marini, Conor Hogan, Myrta Grüning, and Daniele Varsano. yambo: An ab initio tool for excited state calculations. *Computer Physics Communications*, 180(8):1392–1403, August 2009.

[2] Francesco Sottile. *Response functions of semiconductors and insulators : from the Bethe-Salpeter equation to time-dependent density functional theory.* PhD thesis, École Polytechnique, 2003.

[3] H.R. Philipp. Optical transitions in crystalline and fused quartz. *Solid State Communications*, 4(1):73–75, 1966.

[4] Andrea Marini, Rodolfo Del Sole, and Angel Rubio. Bound excitons in time-dependent density-functional theory: Optical and energy-loss spectra. *Phys. Rev. Lett.*, 91:256402, December 2003.

[5] Huabing Shu, XiangHong Niu, XiaoJin Ding, and Ying Wang. Effects of strain and surface modification on stability, electronic and optical properties of GaN monolayer. *Applied Surface Science*, 479:475–481, 2019.

[6] Thomas Galvani, Fulvio Paleari, Henrique P. C. Miranda, Alejandro Molina-Sánchez, Ludger Wirtz, Sylvain Latil, Hakim Amara, and Fran çois Ducastelle. Excitons in boron nitride single layer. *Phys. Rev. B*, 94:125303, September 2016.

[7] Davide Sangalli, Andrea Ferretti, Henrique Miranda, Claudio Attaccalite, Ivan Marri, Elena Cannuccia, Pedro Melo, Margherita Marsili, Fulvio Paleari, Antimo Marrazzo, Gianluca Prandini, Pietro Bonfà, Michael Atambo, Fabio Affinito, Maurizia Palummo, Alejandro Molina-Sánchez, Conor Hogan, Myrta Grüning, Daniele Varsano, and Andrea Marini. Many-body perturbation theory calculations using the yambo code. *Journal of Physics: Condensed Matter*, 31, April 2019.

[8] Jai Singh. *Theory of Excitons*, pages 1–45. Springer US, Boston, MA, 1994.

[9] E. E. Salpeter and H. A. Bethe. A relativistic equation for bound-state problems. *Phys. Rev.*, 84:1232–1242, December 1951.

[10] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[11] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, USA, 1994.

[12] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue*, 6(2):40–53, March 2008.

[13] Daniel Wortmann, Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, et al. Third

release of max software: Report on the release of documentation of the performance optimised parts. Technical report. http://www.max-centre.eu/sites/default/files/D2.3%20Third%20release%20of%20MaX%20software_report%20on%20the%20release%20of%20documentation%20of%20the%20performance%20optimised%20parts.pdf.

[14] James Hardy Wilkinson. *The algebraic eigenvalue problem*, volume 662. Oxford Clarendon, 1965.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[16] Inderjit S Dhillon, Beresford N Parlett, and Christof Vömel. The design and implementation of the MRRR algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 32(4):533–560, 2006.

[17] Jan JM Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36:177–195, 1980.

[18] Christof Vömel, Stanimire Tomov, and Jack Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal on Scientific Computing*, 34(2):C70–C82, 2012.

[19] Michael T Heath. *Scientific computing: an introductory survey, revised second edition*. SIAM, 2018.

[20] G. W. Stewart. A Krylov–Schur Algorithm for Large Eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 23(3):601–614, 2002.

[21] Vicente Hernández, Jose E Román, Andrés Tomás, and Vicente Vidal. Krylov-schur methods in slepc. Technical report, 2007. https://slepc.upv.es/documentation/reports/str7.pdf.

[22] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005.

[23] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. https://petsc.org/, 2023.

[24] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc/TAO users manual. Technical Report ANL-21/39 - Revision 3.19, Argonne National Laboratory, 2023.

[25] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[26] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[27] Burton S Garbow. EISPACK—A package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179–184, 1974.

[28] 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, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[29] A Marek, V Blum, R Johanni, V Havu, B Lang, T Auckenthaler, A Heinecke, H-J Bungartz, and H Lederer. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, May 2014.

[30] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*, 2(3):272–278, 2011. Social Computational Systems.

[31] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[32] Yuchen Ma, Michael Rohlfing, and Carla Molteni. Excited states of biological chromophores studied using many-body perturbation theory: Effects of resonant-antiresonant coupling and dynamical screening. *Phys. Rev. B*, 80:241405, December 2009.

[33] Sahar Sharifzadeh. Many-body perturbation theory for understanding optical excitations in organic molecules and solids. *Journal of Physics: Condensed Matter*, 30(15):153002, March 2018.

[34] Peter Puschnig and Claudia Ambrosch-Draxl. Going beyond the Tamm-Dancoff approximation in the Bethe-Salpeter approach to the optical properties of solids. January 2007. `https://www.researchgate.net/publication/253356477_Going_beyond_the_Tamm-Dancoff_approximation_in_the_Bethe-Salpeter_approach_to_the_optical_properties_of_solids`.

[35] D Sangalli, A Ferretti, H Miranda, C Attaccalite, I Marri, E Cannuccia, P Melo, M Marsili, F Paleari, A Marrazzo, G Prandini, P Bonfà, M O Atambo, F Affinito, M Palummo, A Molina-Sánchez, C Hogan, M Grüning, D Varsano, and A Marini. Many-body perturbation theory calculations using the yambo code. *Journal of Physics: Condensed Matter*, 31(32):325902, May 2019.

[36] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Alp Dener, Matthew Knepley, Scott E. Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Computing*, 108:102831, December 2021.

[37] Meiyue Shao, Felipe H. da Jornada, Chao Yang, Jack Deslippe, and Steven G. Louie. Structure preserving parallel algorithms for solving the Bethe–Salpeter eigenvalue problem. *Linear Algebra and its Applications*, 488:148–167, 2016.

[38] Meiyue Shao and Chao Yang. Bethe-Salpeter Eigenvalue Solver Package (BSEPACK) v0.1, 2017.

[39] Carolin Penke, Andreas Marek, Christian Vorwerk, Claudia Draxl, and Peter Benner. High performance solution of skew-symmetric eigenvalue problems with applications in solving the Bethe-Salpeter eigenvalue problem. *Parallel Computing*, 96:102639, 2020.