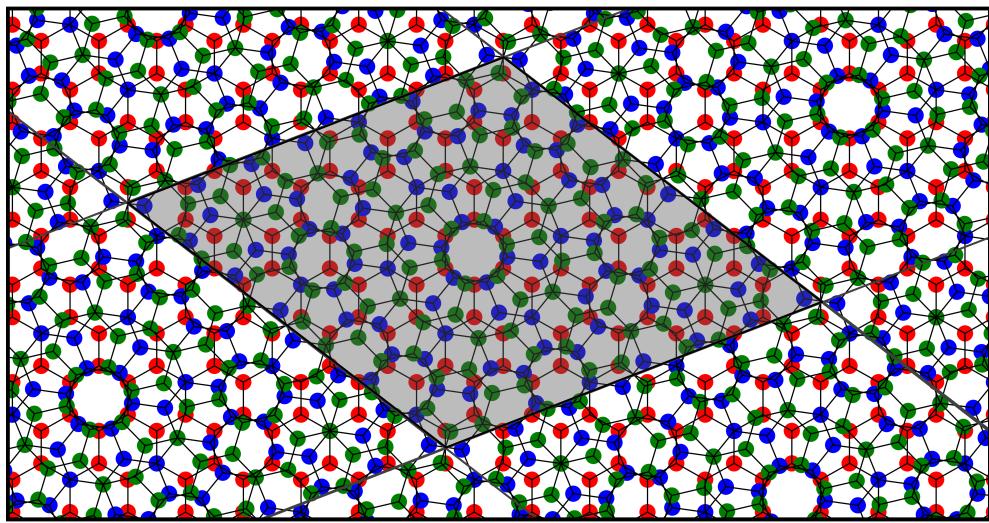


Nook'iin Overview

Ossiel Aguilar-Spíndola



2024

Index

Index	i
I English Version	1
1 Introduction	2
2 General Structure of the Program	8
2.1 Basics.py	8
2.2 Atom.py	9
2.3 Lattice.py	9
2.4 Functions.py	10
2.5 System.py	10
2.6 Interface.py	10
3 Main Algorithm	11
4 Essential Methods and Algorithms	19
4.1 Candidate Search for Primitive Vectors	19
4.1.1 Determining \mathbf{u}_L for a \mathbf{u}	19
4.1.2 Error Calculation for Translation Vectors	21
4.1.3 Algorithm	24
4.2 Transformation Matrix Calculation	26
4.3 Deformation Matrix Calculation	29
4.4 Calculation and Display of the First Brillouin Zone	33
Appendix A Functions	41
A.1 Basics	41
A.1.1 Vector Operations	41
A.1.2 Matrix Operations	42
A.1.3 Auxiliary Functions	43
A.2 Atom	45

A.2.1 Initialization	45
A.2.2 Atom Methods	45
A.3 Lattice	46
A.3.1 Initialization	46
A.3.2 Lattice Methods	47
A.4 Functions	50
A.4.1 Functions on Atoms in an Atomic Basis	50
A.4.2 Primitive Cell Calculation for a Network	50
A.4.3 Transformation Matrix Data Handling	51
A.4.4 Exporting a Network from a POSCAR File	52
A.4.5 Predefined Lattices	52
A.5 System	53
A.5.1 Initialization	53
A.5.2 System Methods	54

II Versión en Español 1

1 Introducción	2
2 Estructura general del programa	9
2.1 Basics.py	10
2.2 Atom.py	10
2.3 Lattice.py	10
2.4 Functions.py	11
2.5 System.py	11
2.6 Interface.py	12
3 Algoritmo principal	13
4 Métodos y algoritmos esenciales	21
4.1 Búsqueda de candidatos a vectores primitivos	21
4.1.1 Determinar \mathbf{u}_L para un \mathbf{u}	21
4.1.2 Cálculo del <i>error</i> para los vectores de traslación	24
4.1.3 Algoritmo	27
4.2 Cálculo de matrices de transformación	30
4.3 Cálculo de la matriz de deformación	33
4.4 Cálculo y muestra de la primera zona de Brillouin	38
Apéndice B Funciones	48
B.1 Basics	48
B.1.1 Operaciones con vectores	48

B.1.2	Operaciones con matrices	49
B.1.3	Funciones auxiliares	50
B.2	Atom	52
B.2.1	Inicialización	52
B.2.2	Métodos de Atom	52
B.3	Lattice	53
B.3.1	Inicialización	53
B.3.2	Métodos de Lattice	54
B.4	Functions	57
B.4.1	Funciones sobre átomos en una Base Atómica	57
B.4.2	Cálculo de la Celda Primitiva de una Red	57
B.4.3	Manejo de datos de Matrices de Trasformación	58
B.4.4	Exportación de una Red desde archivo POSCAR	59
B.4.5	Redes prediseñadas	59
B.5	System	61
B.5.1	Inicialización	61
B.5.2	Métodos de System	62
	Bibliografía	65

Part I

English Version

Introduction

In nature, there are atomic-scale systems¹ with a surface-to-volume ratio of $\geq 10^3$, which allows them to be classified as two-dimensional (2D) materials. One of the most well-known and widely studied 2D materials since its mechanical exfoliation in 2004 by Geim and Novoselov is graphene [1]. Graphene is a system composed of carbon atoms arranged in a hexagonal honeycomb lattice that is only one atom thick. Graphene can be found as a vertically stacked atomic sheet forming graphite, which is what we find in pencil leads, Figure 1.1(a). It is worth noting that the forces that hold graphene sheets together are much weaker than the forces that bind the atoms within each plane.²

As a consequence of Geim and Novoselov's work, theoretical and/or experimental studies (or their combination) of other 2D materials have gained notable interest in the scientific community. Since systems like graphite are composed of weakly bound graphene layers, it is feasible to consider that other materials such as molybdenum disulfide (MoS_2) may also be used to construct vertically stacked heterogeneous systems composed of two or more layers with distinct physical properties, thus creating what are known as heterostructures. These heterostructures hold promising potential for applications in electronics and optoelectronics [2, 3].

The formation of these heterostructures is again possible thanks to the weak interlayer interactions, which are significantly weaker than the intralayer atomic bonds. Specifically, these weak forces are of van der Waals (vdW) dispersive type.³ These systems are known as homo- and hetero- van der Waals structures. The stacking of these layers can be likened to assembling LEGO bricks, where atomic sheets are stacked layer by layer as building blocks, see Figure 1.1(b).

Through vertical stacking of 2D materials, new structures can be created that, relative to the individual physical properties of the constituent materials, may preserve, enhance, or even exhibit entirely new characteristics not observed in the isolated mate-

¹On the scale of 10^{-10} meters.

²The interlayer forces are dispersive in nature, while the in-plane forces are chemical bonds.

³Dipole-dipole interactions are the most representative.

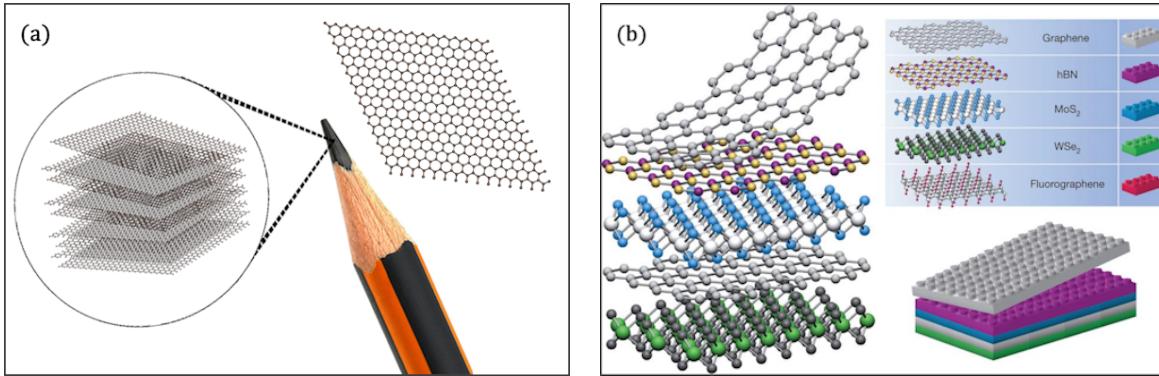


Figure 1.1: (a) Graphene extracted from graphite used in pencils. (b) Right panel: Atomic layers that can be vertically stacked, where each colored sphere represents a different chemical species. Left panel: Analogy between atomic layered systems and LEGO bricks with corresponding vertical stacking. Image taken from the article "*Van der Waals heterostructures*" [4].

rials. Furthermore, systems formed by layers of the same material can display different properties if there are variations in stacking order or rotational alignment between layers. This endows vertical stacking of 2D materials with vast potential for designing materials with diverse physical properties, making such structures of great interest in Physics and Nanoscience, with possible applications in Nanotechnology [5].

Examples of current uses of van der Waals heterostructures include:

- Semiconductor manufacturing, which has contributed to the miniaturization of integrated circuits by replacing beryllium wire connections with aluminum layers and enhancing functionality using composite semiconductors [6, 7].
- Wastewater treatment via photocatalysis using energy band properties [8].
- Development of high-sensitivity photodetectors capable of operating across a wide spectral range, from visible to infrared [3].
- Creation of ultrathin, flexible lasers with promising applications in displays, optical communications, and augmented reality technologies [9].
- Manufacturing of ultrathin LEDs and solar cells for portable and flexible technologies, a growing trend in the industry [10].

As previously mentioned, the thickness of 2D materials is on the order of angstroms (\AA)⁴, and their physical interactions occur at the atomic scale, placing their study within the field of Solid State Physics. This field primarily focuses on the physical properties of crystalline structures, relying on the fact that such structures exhibit periodicity in real

⁴1 \AA = 0.1 nm = $1 \cdot 10^{-10} \text{ m}$

space—whether in one (1D), two (2D), or three dimensions (3D) [11] (Figure 1.2). Based on their periodicity, these crystalline structures, simply referred to as *crystals*, can be described by a **primitive cell**, composed of two key elements: a *Bravais lattice* and an *atomic basis*.

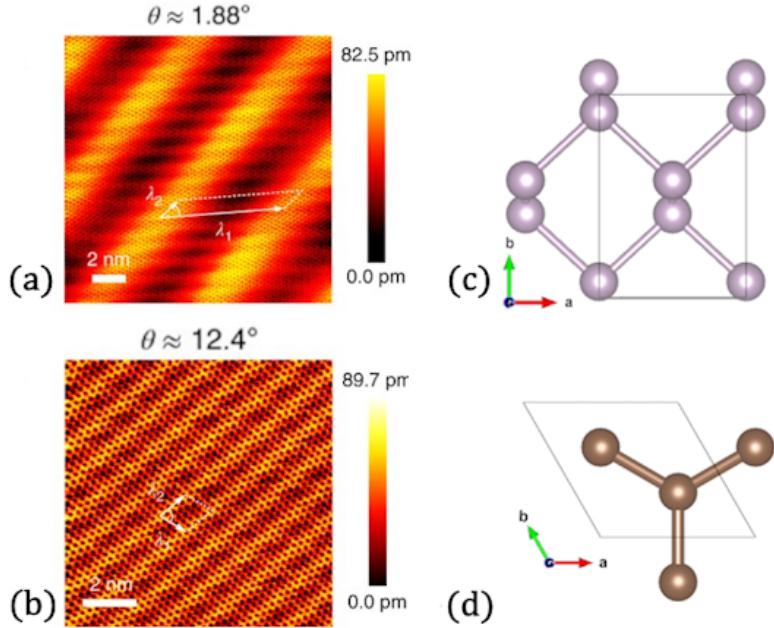


Figure 1.2: STM images of heterostructures formed by two layers of black phosphorene and one of graphene with a rotation of (a) 1.88° and (b) 12.4° , where their periodicity can be observed [12]. Graphical representation of unit cells of (c) black phosphorene and (d) graphene.

The Bravais lattice represents the translational repetition of atoms in real space, while the atomic basis describes the specific arrangement of atoms within the cell (Figure 1.3). The quantum nature of these systems and their atomic-scale characteristics make the determination and understanding of their primitive cells essential for studying physical and chemical properties such as electronic structure and chemical reactivity.

Currently, for the theoretical study of these systems—structures composed of two or more 2D materials—computational tools such as *Density Functional Theory* (DFT)[13] are used. In these cases, it is necessary to identify and characterize the smallest “*building block*” of the system, the primitive cell. Once this is done, if the system is to be studied under perturbations, a larger periodic system must be constructed from the primitive cell, known in the context of Electronic Structure and Solid State Physics as a *supercell*.

The determination of these supercells is not a trivial problem, despite the visualization of these systems as stacked LEGO pieces in Figure 1.1(b). This complexity arises

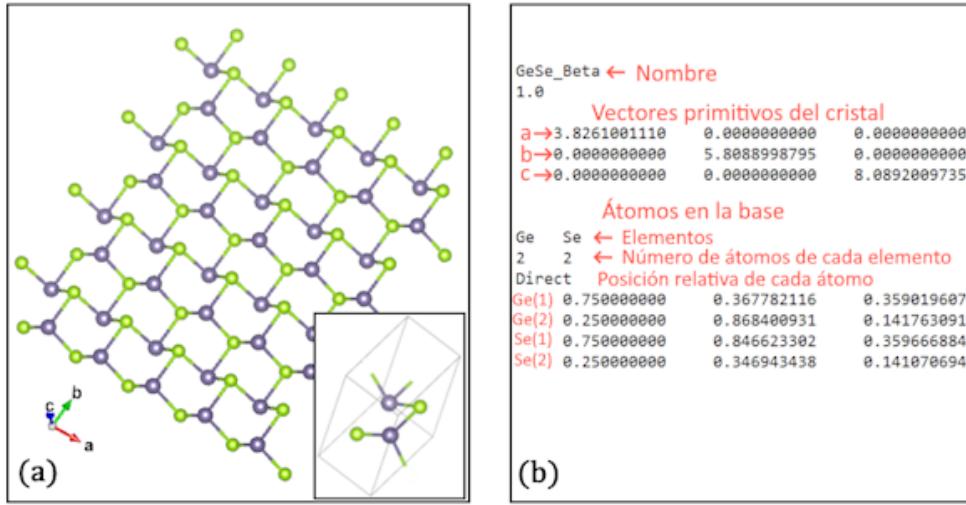


Figure 1.3: (a) Sheet of beta-phase Germanium Selenide (β -GeSe) and its primitive cell. (b) Image of a POSCAR file showing the Bravais lattice vectors and atomic basis of β -GeSe.

because the primitive cell of each layer in the system can differ in shape, size, and orientation. Thus, defining a common supercell that maintains the same periodicity in all atomic layers is challenging, often requiring thousands of atoms. It is important to note that under certain conditions, a common supercell in a system may not be achievable unless tension is applied to the layers forming it, resulting in a new system approximating the original one, but with a supercell that preserves the periodicity of all layers in real space, as illustrated in Figure 1.4.

For all these reasons, obtaining a minimal cell for a crystalline network capable of describing structures composed of two or more layers of 2D materials is the first major problem to solve before studying the physical properties of such structures. There are currently published implementations that address this problem, including *CellMatch* [14], *Supercell-core software* [15], and *Twister* [16].

However, despite these tools attempting to solve the problem, some complications persist. For one, the *CellMatch* implementation uses brute force to find its solution, which negatively affects its computational efficiency. Moreover, this implementation works only with bilayer structures⁵.

On the other hand, the *Twister* implementation primarily addresses the problem of obtaining optimal rotation angles, exclusively for two-layer systems. This limits the types of systems that can be solved and places less emphasis on obtaining the supercell for the

⁵At the start of this work, the URL referenced in the article was a "broken link," preventing access to its code and thus its use or review. However, upon a later review, the link was valid, allowing for analysis.

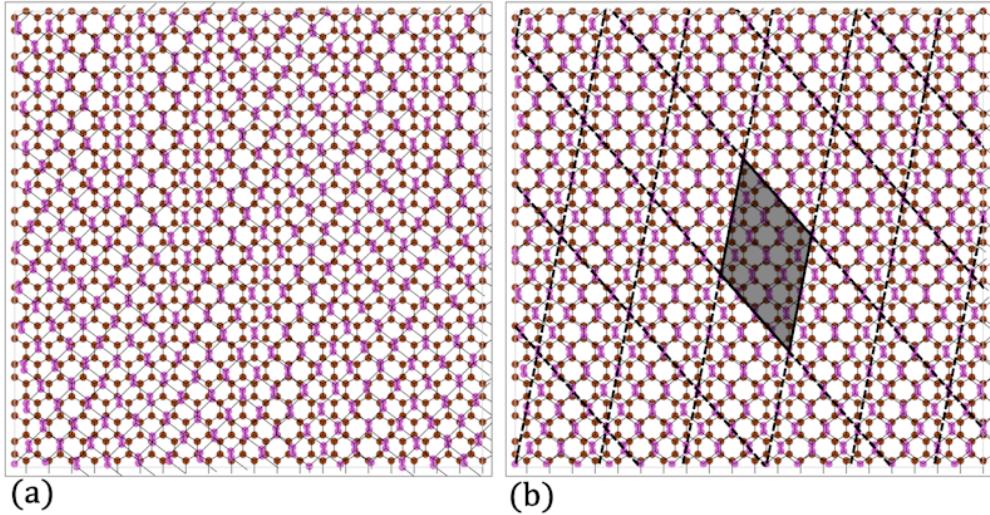


Figure 1.4: Bilayer system of graphene-black phosphorene with a relative interlayer angle of 3.5° . In (a), the system is shown without layer alteration, and no supercell is found that maintains complete periodicity within the observed area. However, in (b), the optimized system is displayed, where the black phosphorene layer underwent deformation, increasing its primitive vectors by -1.94% and $+1.90\%$, respectively. Here, a supercell exists that maintains periodicity throughout the system.

system. Consequently, its code employs a brute-force algorithm, resulting in a solution with complexity $O(n^4)$, where n represents the search space limit for the supercell.

Finally, *Supercell-Core* is a more comprehensive and robust program than the previous two. However, while its results are reliable for specific systems, it still encounters problems in complex systems featuring large supercells. These issues include missing or extra atoms in its atomic basis, possibly due to rounding errors in calculations or deformation methods used. These issues can be difficult to detect in large systems, potentially leading to errors.

Thus, the general objective of this project, as a Bachelor's Thesis, is to develop an open-source computational code (*open source*)^[17] in Python^[18] that is flexible, robust, and intuitive for users. This code will be based on geometric methods^[19, 20] for determining a small, commensurate 2D supercell in systems composed of two or more atomic layers with different relative orientations and chirality, ensuring bounded tension in each layer.

The output of the code will include the lattice vectors in their matrix representation, as well as atomic positions in coordinates relative to the lattice vectors, with the option to export these results to a POSCAR file⁶. This information can be used in first-principles

⁶A POSCAR file is a standard format for describing a material's crystalline structure in terms of lattice

studies to characterize the electronic structure of the system or for potential structural optimizations using molecular dynamics tools with high-performance computing resources.

With this code, in addition to addressing the limitations observed in previous works, information about the reciprocal space corresponding to the computed systems will also be provided—a feature not considered in any of the previously mentioned implementations.

Understanding the image of a crystal in reciprocal space is essential in solid-state physics and electronic structure theory at the atomic scale. Reciprocal space is crucial for comprehending wave dispersion and diffraction phenomena, such as those observed in X-ray diffraction and high-resolution transmission electron microscopy (HRTEM). These phenomena allow for the detailed characterization of a material's crystalline structure and atomic arrangement.

Additionally, reciprocal space imagery provides information on the periodicity and symmetry of the crystalline lattice—key aspects for understanding material properties. For example, peaks in an X-ray diffraction pattern or an HRTEM image correspond to reciprocal lattice vectors related to the material's crystal structure. These peaks can provide insights into the crystalline structure, crystal orientations, and lattice distortions.

In summary, the additional information provided by the program is crucial for characterizing the structure and properties of materials at the atomic level, with significant implications in fields such as materials science, nanotechnology, and electronics.

The interdisciplinary approach of this thesis combines concepts from materials physics, crystallography, programming, and computational simulation, enabling the analysis of complex systems using advanced methods. Additionally, the code being open-source is particularly important, as it allows other researchers to use and contribute to the development of both the software and their own scientific projects.

vectors and atomic positions within a unit cell.

General Structure of the Program

The program presented here is composed of six Python files: *Basics.py*, *Atom.py*, *Lattice.py*, *Functions.py*, *System.py*, and *Interface.py*. Each of these files is imported by the next one, creating a sort of encapsulation (Figure 2.1). This structure ensures that the functions defined in each file are used by the subsequent ones, maintaining order and hierarchy. The user interacts only with the outermost layer, *Interface.py*, which guides them step by step through the console to create and analyze a system.¹ A brief description of each file is provided below.

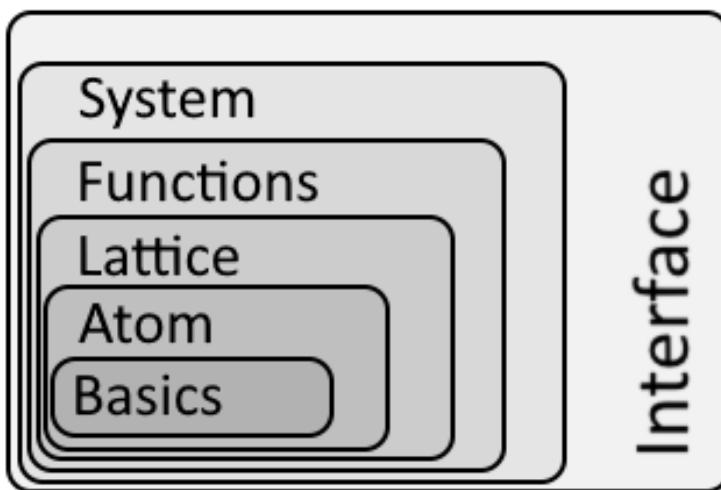


Figure 2.1: Schematic representation of the general structure of the program. This figure illustrates how each layer encapsulates the previous one until reaching the user interface.

2.1 Basics.py

This file contains essential functions that form the foundation of the program. The functions can be categorized into three groups: vector operations, matrix operations, and

¹If the user prefers to use the program tools manually, they can simply import the *System.py* file to access all functionalities directly.

miscellaneous functions. The latter includes methods for solving specific problems, such as finding nearest neighbor lattice points or calculating the vertices of a Wigner-Seitz cell.

These basic functions are written manually to ensure independence from external libraries beyond Python's standard library, which enhances the program's portability and removes the need for third-party package installation. It also provides better control over the internal operations of the program.

2.2 Atom.py

This file defines the class `Atom`, which serves as a blueprint for creating the atomic objects used in the program to model the atoms forming the crystal's atomic basis.

This class is quite basic and handles only the minimum atomic information required by the program. Although it omits various modeling features, it suffices for the program's needs.

The strength of the `Atom` class lies in its ability to represent and classify atoms with simplicity while providing a consistent and organized structure for building more complex systems.

2.3 Lattice.py

This class is used to create objects that serve as integral representations of the crystals in the program. They are defined by their primitive lattice vectors and an atomic basis supplied as a list of `Atom` objects.

In addition to initializing `Lattice` objects, this class includes functions that operate directly on them. These functions play a crucial role in manipulating and analyzing crystal structures.

This class includes functions ranging from modifying the object itself to exporting crystal data to a POSCAR file or displaying a visual representation on screen.

2.4 Functions.py

The `functions.py` file contains several functions that primarily operate through interactions between `Lattice` objects. The main purpose of these functions is to find common supercells of similar dimensions across all objects in the list and determine the adjustments needed to match them without exceeding a defined deformation threshold.

Besides these, the file also includes functions that quickly generate hexagonal and rectangular `Lattice` objects, where only the periodicity size(s) (in Å) are required as input.

There are also functions for initializing predefined `Lattices` for recurring materials such as *graphene*, *black phosphorene*, and the α and β phases of the GeSe compound, denoted for simplicity as α -GeSe and β -GeSe.

2.5 System.py

This file defines the `System` class, which creates objects that model the heterostructures under study. The core of these objects is a list of `Lattice` objects representing the vertically stacked materials that form the modeled heterostructure.

The class includes functions that operate on the `System` object, such as finding possible primitive cells, analyzing the obtained results, and presenting them to the user in a way that supports decision-making based on the specific problem being addressed.

2.6 Interface.py

This file is the program's outermost layer. It interacts directly with the user, providing access to the creation and manipulation of `System` objects to model structures whose primitive cells are to be determined.

Main Algorithm

In the analysis of the presented software, the structure under study is modeled as S , an object of the `System` class defined by a list of `Lattice` objects.¹ These objects represent the 2D periodic lattices corresponding to the layers stacked to form the heterostructure. Once the primitive cell for S is determined, a new `Lattice` object called `SuperRed` is defined and associated with S ($S.SuperRed$), allowing S to be seen as a 2D periodic lattice.

Internally, the program identifies the first element in the list of lattices that define S ($S.Layers[0]$) as the **substrate layer**, which is the layer used as a reference since it will not undergo deformation during the solution process.

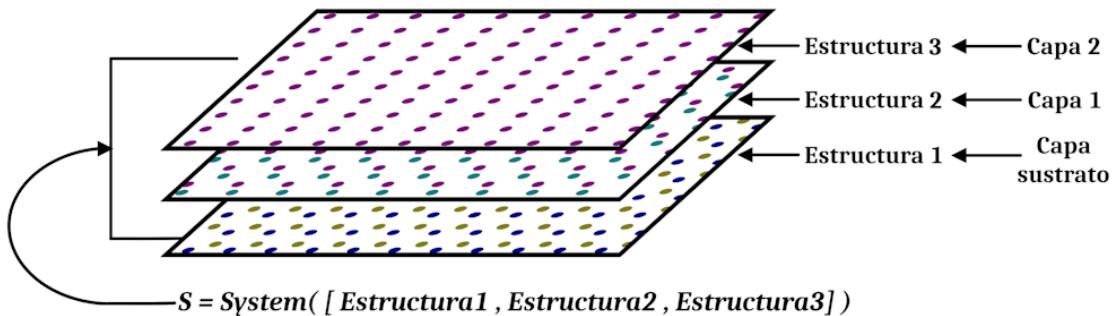


Figure 3.1: General representation of how the software abstracts the system from the list of initial structures.

As mentioned earlier, if supercells for each layer's lattice can be found that coincide when projected onto the plane, a potential primitive cell for the system has been identified (see Figure 3.2). Conversely, a cell that serves as a valid primitive cell for system S will coincide with a supercell of each individual lattice in the system. Furthermore, the vectors describing these supercells will be valid translation vectors for each lattice.

This property implies that the primitive vectors of the system must align with a translation vector of each lattice. Therefore, the primitive cells of the system can be related to

¹Both `System` and `Lattice` classes will be discussed in detail later in this chapter.

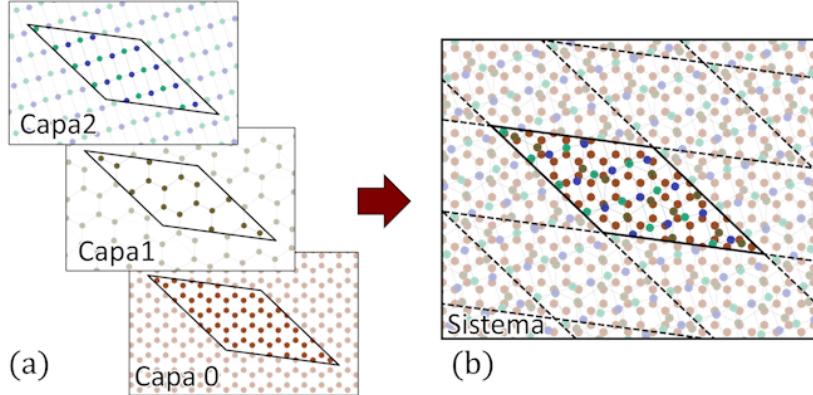


Figure 3.2: Panel (a) shows supercells of three distinct crystalline lattices from different layers of a stacked system, which coincide when projected onto the plane. Panel (b) presents a primitive cell for the heterostructure, matching the stacking of the supercells in (a).

a transformation matrix (TM) of one of its layers. In the program, this is defined in terms of the TM of the substrate layer, since, as noted earlier, it remains unaltered throughout the process.

In 2D structures, a TM is a 2×2 matrix composed of integer coefficients of the translation vectors describing the supercell linked to the TM (an example is shown in Figure 3.3). This matrix is constructed as follows: Let \mathbf{a} and \mathbf{b} be the primitive vectors of a 2D lattice L . The TM corresponding to the supercell formed by $\mathbf{s}_a = (m\mathbf{a} + n\mathbf{b})$ and $\mathbf{s}_b = (p\mathbf{a} + q\mathbf{b})$ is:

$$T = \begin{pmatrix} m & p \\ n & q \end{pmatrix}$$

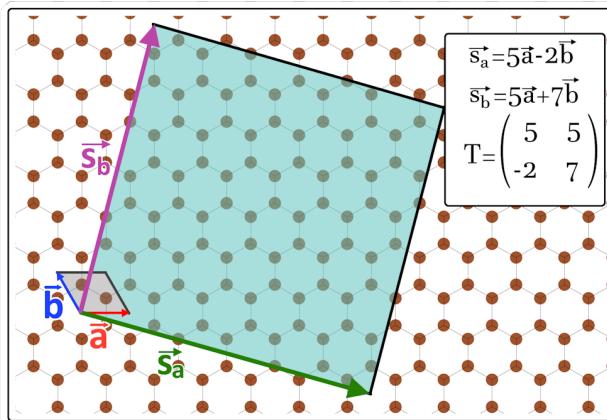


Figure 3.3: TM, T , corresponding to the square supercell defined by translation vectors \mathbf{s}_a and \mathbf{s}_b in a hexagonal lattice with primitive vectors \mathbf{a} and \mathbf{b} .

Given this, the first step in solving the main problem is to identify common translation vectors (TVs) across all layers that can serve as possible primitive vectors for the system. To find these vectors, the program requires additional parameters. First, the search space must be delimited using an integer n , which defines the search area as the parallelogram formed by the substrate layer's supercell spanning from $-n$ to n in both primitive directions \mathbf{a} and \mathbf{b} , as illustrated in Figure 3.4.

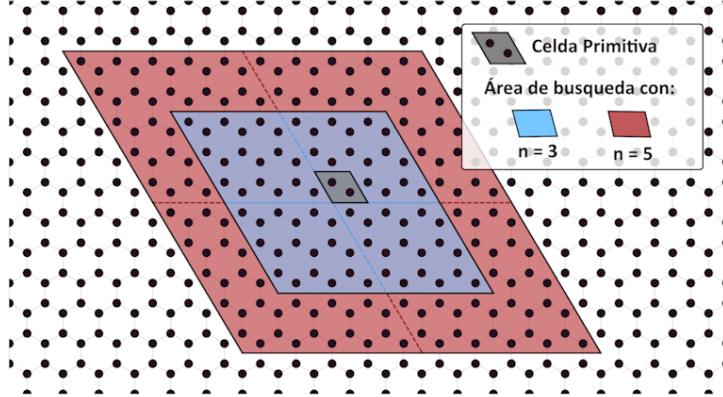


Figure 3.4: In red and blue, search areas for $n = [-3, 3]$ and $n = [-5, 5]$ in a hexagonal lattice. The search area grows quadratically with respect to n .

The second parameter is a constant ε that defines the maximum allowable error between similar translation vectors in all layers. Smaller values of ε result in solutions with less mechanical strain (compression or stretching), though they may require a larger search area to succeed.

In the algorithm, potential primitive vectors for S are derived from a list of lattice points (LPs), R , obtained from the LPs of the substrate layer within the defined search area. This list is "cleaned" to retain only those LPs that exhibit an error less than or equal to ε with corresponding LPs in each layer. This produces a list of LPs shared across all layers within the allowed error.²

Subsequently, potential TMs are generated from R to define the substrate layer's supercell and, in turn, the primitive cell for S . These matrices are built by pairing translation vectors from R , ensuring they produce valid TMs.

To be valid, the resulting matrix must yield linearly independent vectors, which is ensured if the determinant is non-zero.

²The specific algorithm is described in detail later in this chapter.

From the computed matrices, a list *LoMat* is created for *S*, stored as *S.LoMat*, containing the smallest corresponding supercells. At this point, the user may manually choose a TM *T* from *LoMat* or allow the program to automatically select the most optimal one.

To aid this decision, the *ShowTMs* method of the *System* class displays a table summarizing characteristics of the primitive cell for each TM in *LoMat*:

- **Lattice:** Name of the layer's lattice.
- **T:** TM defining the supercell for the primitive cell.
- **Deformation:** Deformation matrix needed to preserve periodicity.
- **Distortion $\delta // \theta$:** Changes in magnitude (δ) and direction (θ) of the primitive vectors.
- **#Atoms:** Number of atoms per layer.

Finally, the total number of atoms in the primitive cell and a DD (Degree of Distortion) score are displayed, representing the overall distortion required to achieve periodicity with the chosen primitive cell.³

Once a suitable TM *T* is selected, the primitive cell for *S* is calculated using the method *S.createSuperLattice(T)*, which creates a new *Lattice* object called *Supercell*. This represents the lattice formed by the primitive cell defined by *T*. The result can be displayed in a 2D projection in real space and reciprocal space using the *show* method (see Figure 3.6).

Additionally, if required, the *exportLattice* function of the *System* class can create a POSCAR file⁴ for the calculated supercell. This file can be used for *ab initio* calculations [21] or visualized with crystallographic software [22].

Analyzing the main algorithm's complexity, we observe it is primarily determined by the routines generating list *R* and list *LoMat*. These depend on the number of lattice points in the substrate layer within the search space, with $O(n^2)$ complexity for generating *R* and $O(n^4)$ for generating *LoMat*.⁵ Consequently, the main algorithm's overall complexity is $O(n^4)$, although in practice it often performs significantly better than this theoretical bound.

³The deformation matrix, δ , θ , and DD are discussed later in this chapter.

⁴This format is used for geometry and composition input in first-principles calculations based on DFT, such as with VASP (Vienna *ab initio* Simulation Package).

⁵These results are discussed in the following section of this chapter.

***Option 1: T <- Matrix loMat[0]					
Lattice	T	Deformation	Distortion: δ/θ	#Atoms	
Grafeno	1 3 -7 -2	1.00000 0.00000 0.00000 1.00000	+0.0% // +0.0° +0.0% // +0.0°	38	
Grafeno(13.15°)	-1 2 -8 -3	1.00024 -0.00047 0.00047 0.99976	+0.0% // +0.02° +0.0% // +0.02°	38	
Black-Phosphorene(27.50°)	1 2 -4 -2	1.07381 0.04535 0.12880 1.07834	+4.129% // -1.53° -2.132% // +2.22°	24	

Total atoms:100 DD: 2.5147

***Option 2: T <- Matrix loMat[1]					
Lattice	T	Deformation	Distortion: δ/θ	#Atoms	
Grafeno	3 9 -2 1	1.00000 0.00000 0.00000 1.00000	+0.0% // +0.0° +0.0% // +0.0°	42	
Grafeno(13.15°)	2 8 -3 -1	0.97872 -0.01482 -0.03541 0.97584	+0.0% // +0.02° +0.0% // -1.51°	44	
Black-Phosphorene(27.50°)	2 6 -2 -2	0.97132 -0.05713 0.03181 0.98134	-2.132% // +2.22° -2.609% // +2.28°	32	

Total atoms:118 DD: 2.6946

Figure 3.5: Tables for TMs *LoMat[0]* and *LoMat[1]* calculated for a heterostructure formed by two graphene layers and one black phosphorene layer. The top graphene layer is rotated 13.15° relative to the bottom one, and phosphorene is rotated 27.5° relative to the same reference. In this case, *LoMat[0]* is recommended due to lower distortion and smaller size (100 atoms).

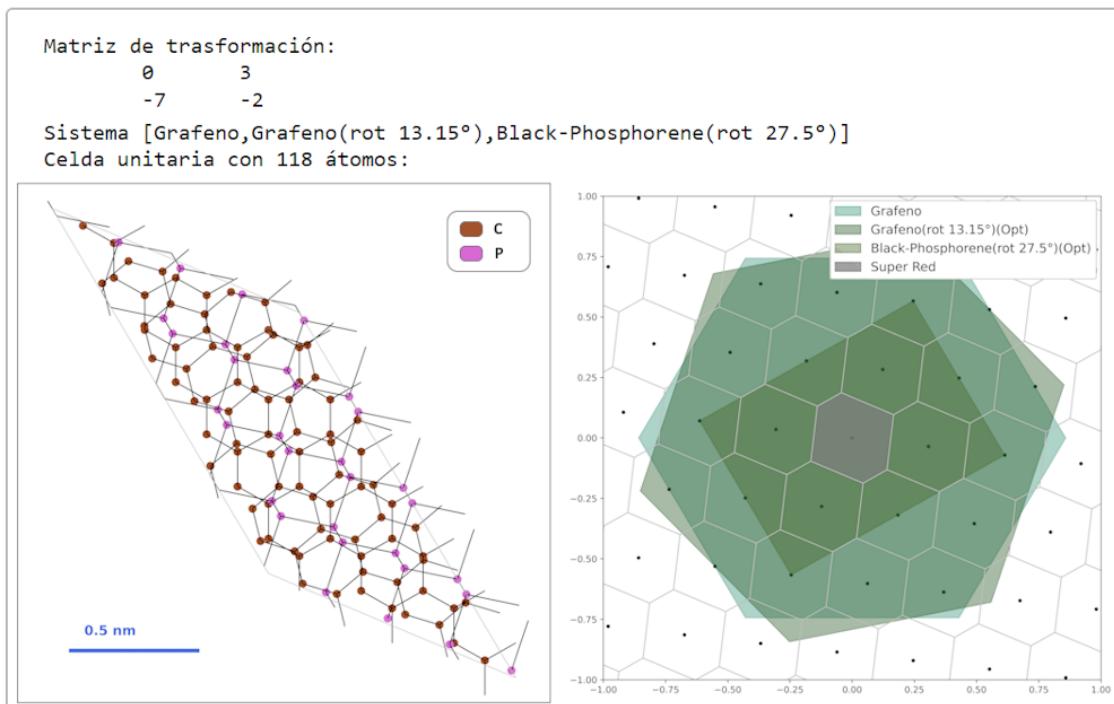


Figure 3.6: Images shown when generating the supercell of a system with two graphene layers and one black phosphorene layer. Left: 2D real-space representation. Right: reciprocal-space view. Reciprocal lattice points are marked in black, surrounded by distorted hexagonal cells like the purple one centered at the origin. The first Brillouin zones of each layer's primitive cell are also shown: two hexagonal (graphene) and one rectangular (phosphorene).

The main algorithm of the program can be summarized as follows:

Main Algorithm	(3.1)
<ol style="list-style-type: none"> 0. Start. 1. Create S. 2. If a transformation matrix T is available, jump to step 11; otherwise, continue. 3. Define n and ε for the search. 4. Calculate list R with translation vectors satisfying ε for all layers of S. 5. If R is empty, continue; otherwise, go to step 7. 6. If new, larger values for n or ε can be used, update and return to step 3. Otherwise, no primitive cell can be found in the search area; terminate (go to 17). 7. Compute list $LoMat$ for S. 8. If $LoMat$ is empty, return to step 6; else continue. 9. If the user wishes to select a T manually from $LoMat$, continue; else go to step 12. 10. Display TM data in table form. 11. Enter T and go to step 13. 12. Select the best T from $LoMat$. 13. Compute the Lattice that will serve as the primitive cell for S. 14. Display the result. 15. If exporting the supercell is desired, continue; else, finish. 16. Create POSCAR file for the supercell. 17. End algorithm. 	

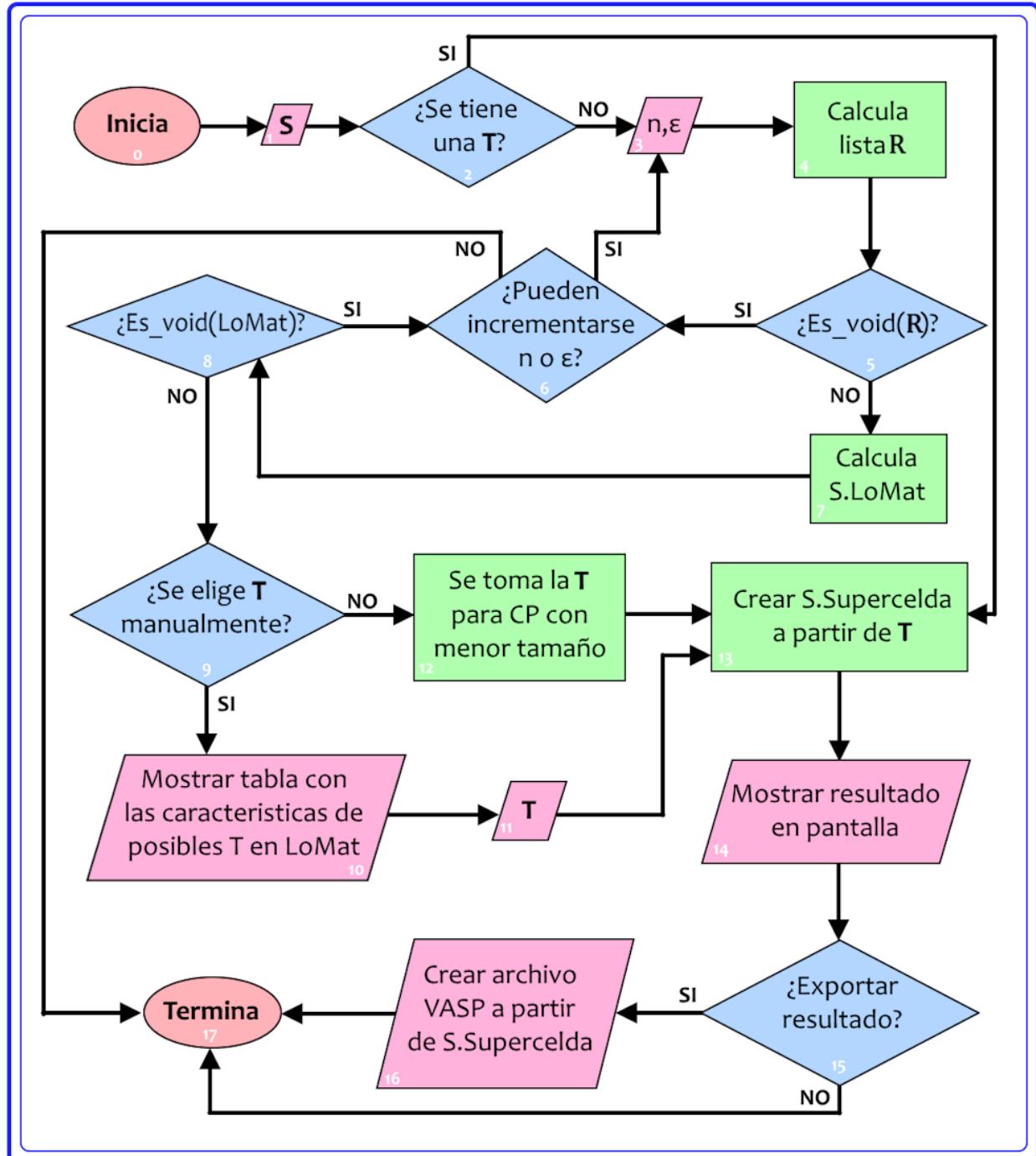


Figure 3.7: Flowchart of the general algorithm used by the program to find the supercell.

Essential Methods and Algorithms

In this section, we address the core algorithms used to solve the problem, as well as the main methods that implement them. These algorithms stand out among all others due to their complexity and significance.

The detailed explanation of these instruction sets will provide a deep understanding of their functionality and the underlying theory that supports them. This analysis aims not only to demonstrate the robustness of these methods but also to shed light on their practical application and relevance in solving the problem for advanced materials research.

4.1 Candidate Search for Primitive Vectors

Following the main algorithm (Algorithm 3.1), if we do not know an MT T that solves the problem, then, once the values n and ε are determined, the first step is to compute a list of possible primitive vectors, R , for the evaluated system, S . These vectors must be within the search area bounded by n and must have a mismatch smaller than ε .

To accomplish this task, two things are required. The first is a method that allows us to find the VT \mathbf{u}_L in a network L that is the closest to the VT \mathbf{u} in another network. The second is a way to measure the difference, mismatch, or error between the original vector and the approximated one. With these two tools, we can obtain the desired list.

4.1.1 Determining \mathbf{u}_L for a \mathbf{u} .

The responsible method for this task in the program is called `calcCD()`, located in the file *Functions.py*.

To begin, the problem involves two networks, S and L . In network S , the vector \mathbf{u} is a valid translation vector, and we need to find the vector \mathbf{u}_L , which is the valid translation

vector in L that is the closest to vector \mathbf{u} . To solve this problem, the first step is to use the concept of lattice points¹. This allows a two-dimensional network to be simplified into a set of periodically arranged points in the plane, eliminating much noise and simplifying the problem.

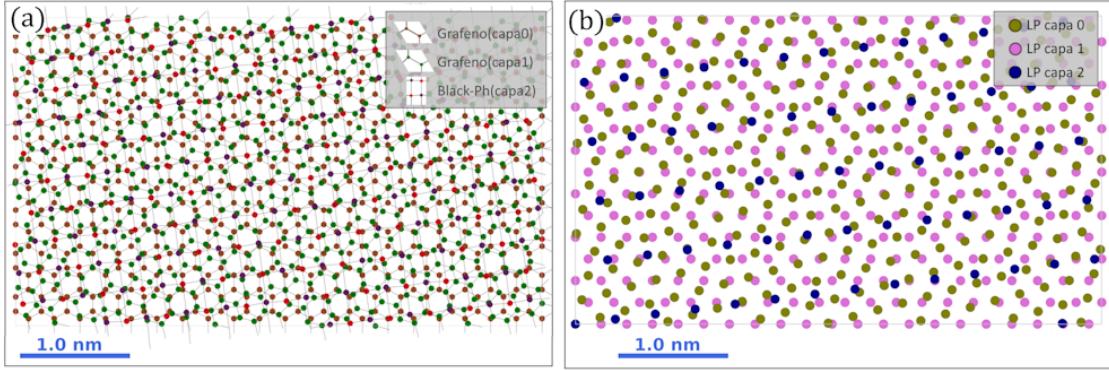


Figure 4.1: (a) Projection in the plane of a system composed of two layers of graphene and one of black phosphorene. The rotation between layer 0 and 1 is 15.5° , while the rotation between layer 0 and 2 is 7.3° . (b) Plane representation of the LPs of the three layers in the composite system.

Let the primitive vectors of a 2D network be \mathbf{a} and \mathbf{b} , with a fixed point in the plane \mathbf{P}_0 as the origin LP. Considering that, due to periodicity, each LP corresponds to the translation of point \mathbf{P}_0 by a VT T such that:

$$T = i\mathbf{a} + j\mathbf{b}$$

we can relate each LP (and the VT that defines it) to a pair of integer values.

$$\mathbf{P}_{i,j} = \mathbf{P}_0 + \mathbf{T}_{i,j} = \mathbf{P}_0 + i\mathbf{a} + j\mathbf{b}, \quad \text{where } i, j \in \mathbb{Z} \quad (4.1)$$

Consequently, the system can be viewed as a set of periodically arranged point families in the plane (one for each layer in the system), where all families contain the point \mathbf{P}_0 . Thus, the problem of finding a common VT between two networks in a system translates into finding a point other than \mathbf{P}_0 that is shared among their respective families.

Given that the primitive vectors of a network U in 2D must be linearly independent, they can be used as a basis β (in this case, for \mathbb{R}^2). It can then be observed that using

¹Discrete points in space that define the periodicity of a crystalline lattice. These points correspond to the positions where the crystal structure's units (such as atoms, ions, or molecules) repeat identically according to a set of translation vectors defined by the unit cell parameters.

β as a basis for the plane, each position $(x, y)_{[\beta]}$, with $x, y \in \mathbb{Z}$, yields the LP $P_{x,y}$ of network U . Exploiting this phenomenon, the following approach is planned. We define the notation $T_{R,i,j}$ as the translation vector of network R with integer components i and j , and let $P_{i,j}$ be the LPs of S and $Q_{k,l}$ be the LPs of L .

If we have a vector $\mathbf{u} = T_{S,a,b}$, we can locate its corresponding LP $P_{a,b}$ in the plane. Assuming $Q_{c,d}$ is the closest LP in L to $P_{a,b}$, then performing a basis change using β (the basis formed by the primitive vectors of L) provides a new position for $P_{a,b}$. Here, $P_{a,b}$ will still be close to $Q_{c,d}$ (since this basis change is not always an *isometric* transformation, this cannot be generalized, but in very small neighborhoods, it can be assumed to hold).

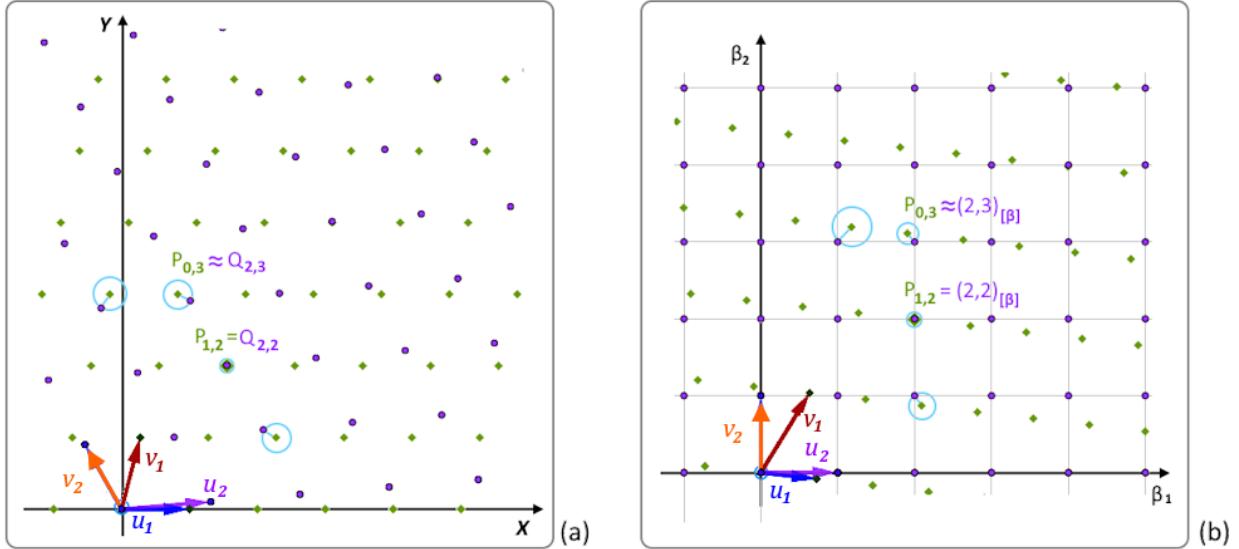


Figure 4.2: (a) Planar projections of LPs for two networks, one with primitive vectors $\mathbf{u}_1, \mathbf{v}_1$ (green diamonds) and another with primitive vectors $\mathbf{u}_2, \mathbf{v}_2$ (purple circles). It shows how, if in (a) the canonical basis for the plane, an LP $P_{a,b}$ is very close to an LP $Q_{c,d}$, then in (b) the plane with basis $\beta = \{\mathbf{u}_2, \mathbf{v}_2\}$, we have $P_{a,b} \approx (c, d)$.

4.1.2 Error Calculation for Translation Vectors

This function plays a fundamental role in the overall functionality of the program, as it is used as the first filter to reach a final result. The main purpose is to establish an evaluation criterion for each candidate vector for VP, in order to discern which are the best options among them.

To select among the VP candidates, it is necessary to define a function, $err(\mathbf{u}, \mathbf{u}_L)$, that evaluates the error $\epsilon_{\mathbf{u}_L}$ of each VT \mathbf{u} in the substrate layer, with respect to a vector \mathbf{u}_L ,

which corresponds to the VT of network L that is most similar to \mathbf{u} . This function must satisfy the following conditions:

- The value of $\varepsilon_{\mathbf{u}_L}$ must be proportional to the deformation required in network L to maintain periodicity throughout the structure.
- For each \mathbf{u} , there must exist a unique value $\varepsilon_{\mathbf{u}_L}$ that qualifies it.
- The value $\varepsilon_{\mathbf{u}_L}$ must always be non-negative.
- If $\varepsilon_{\mathbf{u}_L} = 0$, then the result is optimal. The higher its value, the less suitable the vector \mathbf{u} is as VP.

The fact that $\varepsilon_{\mathbf{u}_L}$ is non-negative and that a score close to zero is the optimal result allows for easy interpretation of the rating of \mathbf{u} . Furthermore, together with the relationship between $\varepsilon_{\mathbf{u}_L}$ and the final deformations required in the system, it enables the introduction of a limit ϵ . This threshold value allows discarding vectors that do not meet the required standards, ensuring the selection of optimal vectors for the CP determination process.

The first requirement for calculating $\varepsilon_{\mathbf{u}_L}$ is to identify the corresponding vector \mathbf{u}_L for \mathbf{u} , which has already been defined. Once this vector is known, a way to measure the difference between them can be established.

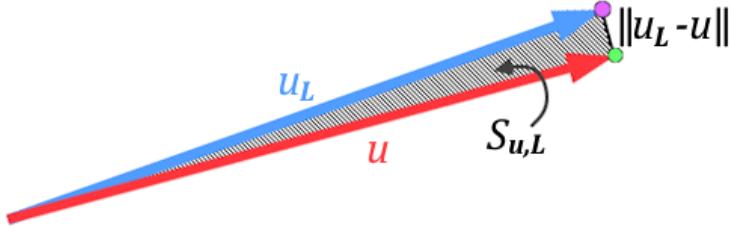


Figure 4.3: $S_{\mathbf{u}_L}$ is the difference between vector \mathbf{u} and its corresponding vector in network L , \mathbf{u}_L .

Knowing \mathbf{u}_L , we can consider that it must be *deformed* until it coincides with \mathbf{u} . This deformation allows us to measure its *error*. Thus, we define $S_{\mathbf{u}_L}$ as the sum of the distances from each point in \mathbf{u}_L to its corresponding point in \mathbf{u} , as illustrated in Figure 4.3.

With this, we can define $err(\mathbf{u}, \mathbf{u}_L) = \varepsilon_{\mathbf{u}_L}$ as the average of $S_{\mathbf{u}_L}$:

$$\varepsilon_{\mathbf{u}_L} = \frac{S_{\mathbf{u}_L}}{\|\mathbf{u}_L\|}. \quad (4.2)$$

Let $\mathbf{u} = (u_x, u_y)$ and $\mathbf{u}_L = (v_x, v_y)$. Then, for each point $\mathbf{v}_t = (t * v_x, t * v_y) \in \mathbf{u}_L$, there corresponds the point $\mathbf{u}_t = (t * u_x, t * u_y) \in \mathbf{u}$ with $t \in [0, 1]$. Thus:

$$S_{\mathbf{u}_L} = \int_0^1 dist(\mathbf{v}_t, \mathbf{u}_t) dt$$

The function “*dist*” returns the distance between two vectors, which is equal to the magnitude of the resulting vector from the input vectors. Given $\mathbf{a} = (a_1, a_2)$ and $\mathbf{b} = (b_1, b_2)$, then:

$$dist(\mathbf{a}, \mathbf{b}) = ||(\mathbf{b} - \mathbf{a})|| = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2}$$

Thus, we have:

$$\begin{aligned} S_{\mathbf{u}_L} &= \int_0^1 \sqrt{(t * u_x - t * v_x)^2 - (t * u_y - t * v_y)^2} dt \\ &= \int_0^1 \sqrt{t^2 ((u_x - v_x)^2 - (u_y - v_y)^2)} dt \\ &= \int_0^1 t \sqrt{(u_x - v_x)^2 - (u_y - v_y)^2} dt \\ &= \int_0^1 t \|\mathbf{u} - \mathbf{u}_L\| dt \\ &= \|\mathbf{u} - \mathbf{u}_L\| \int_0^1 t dt \end{aligned}$$

Thus:

$$S_{\mathbf{u}_L} = \frac{\|\mathbf{u} - \mathbf{u}_L\|}{2} \quad (4.3)$$

Substituting $S_{\mathbf{u}_L}$ from Equation (4.3) into Equation (4.2), we obtain:

$$err(\mathbf{u}, \mathbf{u}_L) = \varepsilon_{\mathbf{u}_L} = \frac{\|\mathbf{u} - \mathbf{u}_L\|}{2\|\mathbf{u}_L\|} \quad (4.4)$$

4.1.3 Algorithm

Having defined the tools used by the program to solve this first step, we now present the following algorithm:

Creation of list R	(4.5)
<pre> 0. Initialize for a system S. 1. Receive n and ε. 2. Initialize the list R with all VTs of the “substrate layer” ($S.networks[0]$) located within the search area. Initialize a counter i starting from 1: $i = 1$. 3. Purify the list R for the network L. Let L be the network in layer i of S, then for each vector $\mathbf{u} \in R$: - Identify \mathbf{u}_L, the VT of L most similar to \mathbf{u}. - Calculate the error between \mathbf{u} and \mathbf{u}_L, $err(\mathbf{u}, \mathbf{u}_L)$. - If the condition $err(\mathbf{u}, \mathbf{u}_L) < \varepsilon$ is not met, then remove \mathbf{u} from R. 4. If $i \geq (len(S.networks) - 1)$, then jump to step 6. 5. Increment the counter i by 1 and return to step 3. 6. Return R as the list of possible VPs for S with the given values for n and ε. 7. Terminate. </pre>	

In the program, `searchLP(rangeOfSearch, epsilon)` is the instruction responsible for executing this calculation, where `rangeOfSearch` corresponds to the value of n and `epsilon` corresponds to the value of ε .

Upon completing the calculation of list **R**, if **R** is empty, it means that no \mathbf{u} exists within the search area among the VTs of the substrate layer of the system. In this case, the corresponding vectors \mathbf{u}_L in all other layers exhibit an error below the given limit. Thus, to obtain a *non-empty* list, it is necessary to either increase the search area by incrementing the value of n , or allow a higher error tolerance by increasing the value of ε .

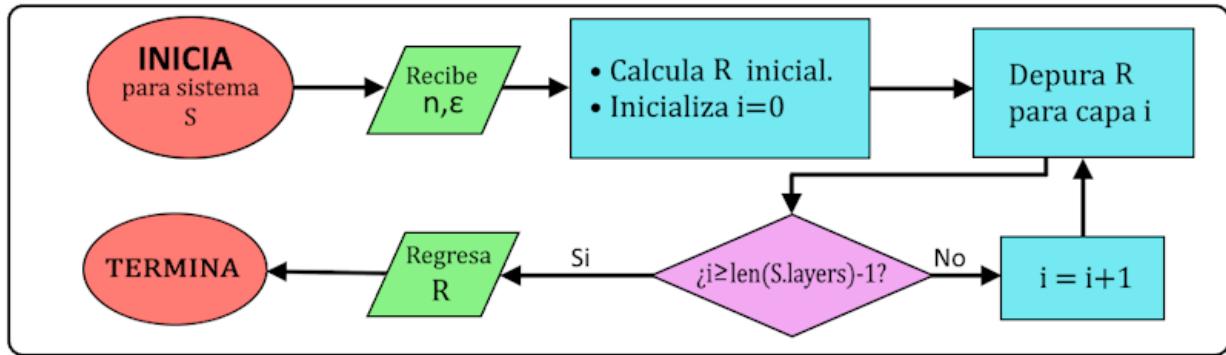


Figure 4.4: Flowchart for the algorithm that calculates VP candidates

When analyzing the complexity of this algorithm, given that the calculation of the vectors u_L for the vectors u and the error computation between them are derived respectively from equations (??) and (4.4), their calculation is performed in constant time ($O(1)$). Thus, it is important to note that the complexity burden of the algorithm is determined by the number of possible vectors u in R and the number of layers in the system.

The size of list R is bounded by the total number of lattice points in layer zero of S (substrate layer) within the search area. This total is $2n \times 2n = 4n^2$. Meanwhile, if we consider the number of layers in the system as c , this is usually a small constant (most evaluated systems will have up to 3 layers), so it can be disregarded, and we can assess this algorithm as belonging to $O(n^2)$, where n refers to the variable indicating the search area for this process.

Furthermore, when conducting a more detailed analysis of the final size of list R , we observe that it can only reach its upper bound if none of the initial PRs are discarded. That is, for all PRs in the substrate layer, there exists a PR in each layer such that its calculated error is below the given ε . This only occurs if the value of ε is very large or if all the networks composing the system have the same orientation and the CP of the substrate layer has the same shape and size as some supercell in each other system layer.

The above scenario arises because, at the stage of the algorithm where comparisons are made against the network of each layer, the initial size of R typically decreases, as there are PRs in the substrate layer whose error with their corresponding PR in that layer exceeds ε . These points are eliminated from list R before proceeding to the next layer's analysis. The magnitude of this reduction is greater the more distinct the networks in the system are and the smaller the given ε . Thus, in systems with rotations and different Bravais networks in each layer, or in analyses with very small ε values, the final size of R will be much smaller than its bound of n^2 .

From the previous analysis, we can also conclude that the final size of list R tends to be inversely proportional to the number of layers in the system.

4.2 Transformation Matrix Calculation

The method responsible for performing this task in the program is `calculateTM()` from the “System” class, which stores the list of calculated matrices as the list *loMat* within the System class.

The TMs indicate a linear transformation that, when applied to the PVs of a network, generate the vectors that form a supercell for that network. Because of this, the determinant of a TM tells us how many times larger the associated supercell is compared to the primitive cell of the network. Therefore, when searching for the smallest supercells to use as primitive cells (PC) of the system, we can compute the determinant of the TMs to sort these results and select the smallest supercells.

Since we already have R , a list of valid VTs across all layers of the system, we can generate TMs by taking pairs of vectors in R . However, not all pairs of vectors yield TMs that generate supercells usable as PCs for the system—for example, if the vectors are collinear. To discard such cases, computing the determinant of the TM is also useful. If the determinant is zero, the TM is discarded, and we proceed with another one.

While computing the determinant, we can also enforce that the vectors retain the same “*orientation*”. Thus, if the determinant is negative, the vectors are swapped so that the new determinant is positive, ensuring coherent and consistent results.

Additionally, since the obtained results are the “smallest”, the number of responses can be limited to reduce computational cost. The limit is set by the class variable *MaxNumM*, which defaults to 10 but can be adjusted by the user. This variable constrains the size of the list *loMat*, restricting the number of TMs displayed to the user per response.

Once the list *loMat* is created, using Eqs. ??, we can obtain the TMs for each layer corresponding to the stored TMs.

The algorithm used to calculate this list of matrices is as follows:

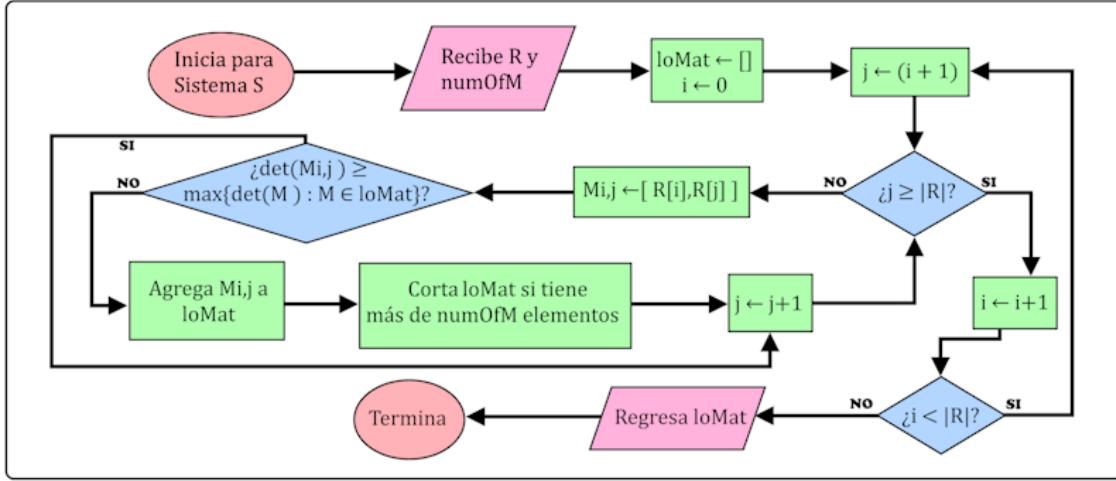
Creation of TM list <i>loMat</i>	(4.6)
----------------------------------	-------

0. Initialize for a system *S*.
1. Receive the list of vectors *R* and the variable *numOfM*.
2. Initialize *loMat* as an empty list. Initialize counter *i* = 0.
3. Initialize counter *j* = *i* + 1.
4. If *j* ≥ |*R*|, then jump to step 10.
5. Create the matrix $M_{i,j}$ from vectors *R*[*i*] and *R*[*j*].
6.

$$\text{biggestM} = \begin{cases} \infty & \text{if } \textit{loMat} \text{ is empty} \\ \max\{\det(M) : M \in \textit{loMat}\} & \text{if } \textit{loMat} \text{ is not empty} \end{cases}$$
 If $\det(M_{i,j}) \geq \text{biggestM}$, jump to step 9.
7. If $M_{i,j}$ is a valid TM and is new to *loMat*, add it in an ordered manner to the list.
8. If $\text{len}(\textit{loMat}) > \text{numOfM}$, trim *loMat* to contain only *numOfM* elements.
9. Increment counter *j* by 1 and return to step 4.
10. Increment counter *i* by 1.
11. If *i* < |*R*|, return to step 3.
12. Return *loMat* as the list of “*numOfM*” TMs that generate the smallest periodic cells for *S*.
13. Terminate.

In step 7 of the algorithm, the computed matrix $M_{i,j}$ is added to the list *loMat* if it is a valid TM and is also new to the list. The first criterion refers to the aforementioned requirement that if the determinant of the matrix is zero, it means the vectors used are collinear. In this case, the matrix is discarded as it does not correspond to a valid cell for the network. The second criterion is slightly more complex but equally important, as it eliminates matrices that are exact rotations of some matrix already included in *loMat*.

Other matrices discarded in the same step are those generating highly unconventional cells, such as those with a very acute angle between the vectors forming them (by default, matrices generating cells with less than 20° of internal angle are discarded), or those where the ratio between the vector sizes is greater than 10.

Figure 4.5: Flowchart for the algorithm that computes the list *loMat*

In analyzing the complexity of this algorithm, we see that the main computational burden is in the process of adding matrices in order by determinant value after evaluation, to the list *loMat*. Thus, the complexity of this algorithm is constrained solely by the number of matrices evaluated, which, for a list *R* of size *m*, results in a number of distinct matrices equal to $\frac{m(m-1)}{2}$. This is because the process of maintaining the order of the resulting list when adding a new matrix is constrained by the constant *numOfM*.

Recalling that the size of list *R* is bounded by the square of the constant *n* that determines the search area for the general algorithm, this algorithm can thus be constrained to $O(n^4)$. However, as previously noted, in execution, the size of list *R* is typically much smaller than n^2 , meaning this bound is far above what is observed when running the code.

Another optimization measure arises in cases where all networks composing the evaluated system are hexagonal networks. In these cases, given the symmetries of the systems, the expected supercell used as the PC should also have a hexagonal Bravais lattice. Thus, calculating all possible pairs of elements from list *R* when creating the matrices to evaluate can be avoided. Consequently, if the system consists solely of hexagonal networks, the following method is used to compute a TM yielding a hexagonal cell for each VT in *R*:

Let $V = x\mathbf{a} + y\mathbf{b}$ be a VT stored in *R*. Then, this vector is associated with the TM, *T*, such that:

$$T = \begin{pmatrix} x & -y \\ y & x - y \end{pmatrix}$$

Thus, when evaluating systems composed solely of hexagonal networks (regardless of differences in lattice constants and/or rotations), the number of matrices evaluated is equal to the size of R .

4.3 Deformation Matrix Calculation

Except for special cases, some layers of the system will need to undergo deformation to maintain complete periodicity. This deformation cannot be too large and is directly limited by the value assigned to the variable ε when computing the VP candidates for the system.

This deformation is reflected in matrices that must multiply the PVs of each layer respectively, ensuring that when the corresponding TM is applied, the results match perfectly with the selected vectors to be used as PVs for the system. This matrix is called the deformation matrix (DM).

Let:

- V_s be the matrix of PVs of the network in the system's substrate layer (layer 0).
- V_i be the matrix of PVs of the network in layer i .
- T be the TM that generates the PC for the system.
- T_i be the TM specific to layer i of the system.

Then the DM for layer i is D_i , a matrix satisfying the following equality:

$$V_s T = (V_i D_i) T_i$$

Since V_i and T_i are matrices with nonzero determinants, they have an inverse, leading to the following equation:

$$D_i = V_i^{-1} (V_s T) T_i^{-1} \quad (4.7)$$

The new PVs for the network of the system in layer i will be obtained from the columns of matrix $V'_i = V_i D_i$, which will generate an adjustment in the PC of the network. The change experienced by a PV \mathbf{u} after deformation can be described more clearly using the variables ΔM_u and ΔA_u , where ΔM_u represents the percentage variation in the magnitude of the vector relative to its original value, and ΔA_u indicates the change in its directional angle.

Let $\mathbf{u} = m (\cos(t)\hat{i} + \sin(t)\hat{j})$ and $\mathbf{u}' = m' (\cos(t')\hat{i} + \sin(t')\hat{j})$ be an original vector and the resulting one after undergoing deformation. We define ΔM_u and ΔA_u as follows:

$$\Delta M_u = \frac{(m' - m)}{m} \times 100 \quad \Delta A_u = (t' - t) \quad (4.8)$$

To quantify the change produced in the network, the following method is used:

Let a, b be the original PVs of the network, and a', b' the deformed PVs. Then, taking the vectors $P_k = i_k a + j_k b$, $Q_k = i_k a' + j_k b'$ with $i_k, j_k \in [0, 1]$, $k \in \{1, 2, \dots, n\}$, we obtain n pairs of vectors extending from the origin to points within the areas delimited by the original and deformed PCs of the network, respectively. The degree of distortion of the network DD is then defined as:

$$DD = \sum_{k=1}^n \left(\frac{\text{err}(P_k, Q_k)}{n} \right) \quad (4.9)$$

Here, the err function defined in Equation (4.4) is used to approximate the average error (the average distortion) generated by the deformation.

Each TM corresponding to a possible solution to the system will induce deformations for each layer of the system (which may be null). These deformations in the networks of each layer can be classified with their respective DD . By construction, the deformation required for the substrate layer is null, always ensuring $DD = 0$ for this layer.

The sum of all DD values across the system layers induced by a TM is termed the total distortion degree (DD_t). This allows us to rate each TM based on the deformation it imposes—lower DD_t values make a TM a better option, with an optimal result being $DD_t = 0$, indicating zero deformation across the system.

After extensive execution tests, we observed that *very good results* are those where DD_t is less than $0.02 * (c - 1)$, with c being the number of layers in the system. This implies that, on average, the “error” associated with the PVs of all layers above the substrate layer will be less than 4%.

Having this “rating” for each result allows the program to optimize both the size of the PC (as the matrices in *loMat* are already the smallest possible TMs given the parameters) and the deformation across the system layers. This is the matrix that the program will recommend as the best answer if the user does not manually select a TM.

To illustrate all of the above, let us consider a system S composed of three layers:

graphene, rotated graphene by 13.5° , and black phosphorene. In this example, we use the TM:

$$T = \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}$$

This TM induces the PVs for the system to be the vectors $\mathbf{u} = 3a_0 - 2b_0$ and $\mathbf{v} = 9a_0 + 2b_0$, where $a_0 = (2.44, 0.0)$ and $b_0 = (-1.22, 2.1131)$ are the PVs of the first graphene layer (the system's substrate layer). Thus:

$$\mathbf{u} = (9.76, -4.2262), \mathbf{v} = (19.52, 4.2262)$$

Calculating the VTs for the rotated graphene network closest to these vectors gives $\mathbf{u}_1 = 2a_1 - 3b_1$ and $\mathbf{v}_1 = 8a_1$. Here, the rotated graphene PVs are $a_1 = (2.3726, 0.5696)$ and $b_1 = (-1.6796, 1.7699)$, yielding:

$$\mathbf{u}_1 = (9.7839, -4.1705), \mathbf{v}_1 = (18.9807, 4.5569)$$

Similarly, for the phosphorene network, we obtain $\mathbf{u}_2 = 3a_2 - b_2$ and $\mathbf{v}_2 = 6a_2 + b_2$. In this case, the network PVs are $a_2 = (3.2601, 0.0)$ and $b_2 = (0.0, 4.347)$, resulting in:

$$\mathbf{u}_2 = (9.7804, -4.3470), \mathbf{v}_2 = (19.5608, 4.3470)$$

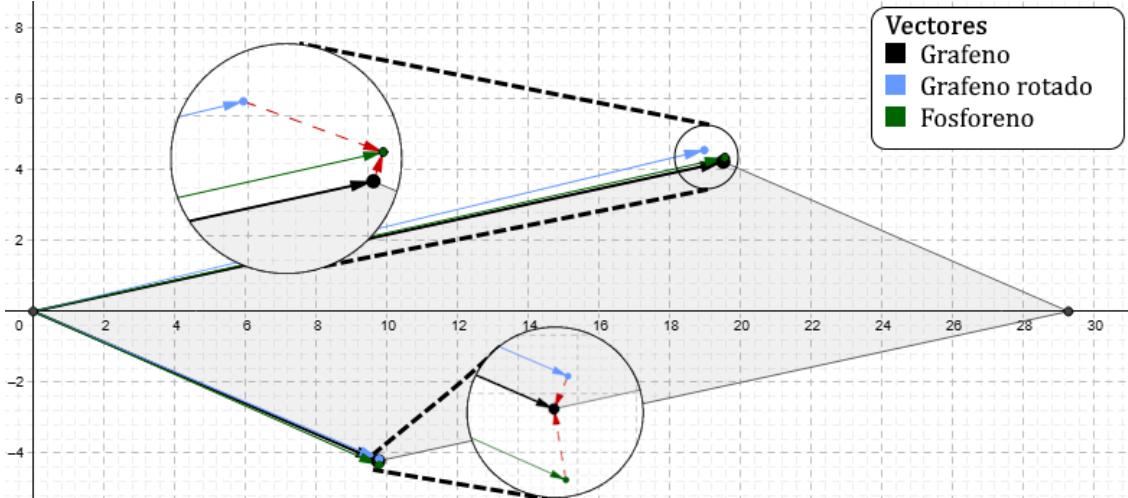


Figure 4.6: Tentative primitive cell for a system composed of graphene, rotated graphene by 13.5° , and black phosphorene. The cell is defined by the VTs of the first graphene layer (black vectors). The closest VTs in the rotated graphene (blue) and phosphorene layers (green) are also shown. It is evident that these do not perfectly coincide with the primitive cell, requiring deformation to align the cells.

From this example, it is clear that $\mathbf{u} \neq \mathbf{u}_1 \neq \mathbf{u}_2$ and $\mathbf{v} \neq \mathbf{v}_1 \neq \mathbf{v}_2$, so if no deformation is applied to the two upper layers, they will lose their periodicity.

For this case:

$$V_s = \begin{pmatrix} 2.4400 & -1.2200 \\ 0.0000 & 2.1131 \end{pmatrix}, T = \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}$$

$$V_1 = \begin{pmatrix} 2.3726 & -1.6796 \\ 0.5696 & 1.7699 \end{pmatrix}, T_1 = \begin{pmatrix} 2 & 8 \\ -3 & 0 \end{pmatrix}$$

$$V_2 = \begin{pmatrix} 3.2601 & 0.0000 \\ 0.0000 & 4.347 \end{pmatrix}, T_2 = \begin{pmatrix} 3 & 6 \\ -1 & 1 \end{pmatrix}$$

Using Equation 4.7, we obtain the corresponding deformation matrices (DMs) for each layer.

$$D_0 = \begin{pmatrix} 2.4400 & -1.2200 \\ 0.0000 & 2.1131 \end{pmatrix}^{-1} \left(\begin{pmatrix} 2.4400 & -1.2200 \\ 0.0000 & 2.1131 \end{pmatrix} \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix} \right) \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}^{-1} \quad (4.10)$$

And similarly for D_1 and D_2 , leading to:

$$D_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, D_1 = \begin{pmatrix} 1.00968 & 0.01524 \\ -0.02647 & 0.99001 \end{pmatrix}, D_2 = \begin{pmatrix} 0.99792 & 0.00000 \\ 0.00000 & 0.97220 \end{pmatrix}$$

This confirms that, except for the system's layer zero, the other layers require deformation. Applying this deformation to each layer's PVs results in new PVs that maintain periodicity across the entire system using the selected PC.

$$V'_1 = \begin{pmatrix} 2.4400 & -1.6267 \\ 0.5283 & 1.7609 \end{pmatrix}, V'_2 = \begin{pmatrix} 3.253318992 & 0 \\ 0 & 4.2261534 \end{pmatrix}$$

These deformations, when applying Equation 4.9 with 100 pairs of vectors, yield $DD_1 = 0.021002$ and $DD_2 = 0.009611$, with a total distortion degree $DDt = 0.030613$. This value indicates that the result is sufficiently accurate, as:

$$DDt < 0.02 * (3 - 1)$$

thus meeting the previously mentioned condition.

Information about the deformation required for system networks for each TM in *loMat*, provided by the method *ShowTMs* mentioned in the previous subsection, includes the corresponding DM for each network, found in the “**Deformation**” column. The ΔM and ΔA values generated by these deformations for their PVs are listed in the “**Distortion δ/ϑ :**” column, where δ corresponds to ΔM and ϑ to ΔA . Finally, the **DD** value expressed in each TM table corresponds to the system's DDt for that TM. These details are displayed in Figure 3.5, located in the “Main Algorithm” section of this chapter, showcasing example tables generated by this method.

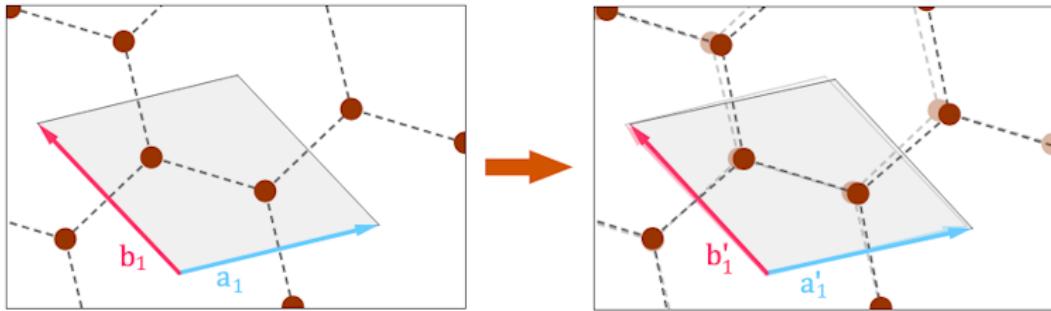


Figure 4.7: Deformation applied to the rotated graphene layer for the defined system. This deformation modifies its PVs a and b such that: $\Delta M_a = 2.3\%$, $\Delta M_b = 1.7\%$, $\Delta A_a = -1.28^\circ$, and $\Delta A_b = -0.76^\circ$. This deformation results in a $DD = 0.021002$.

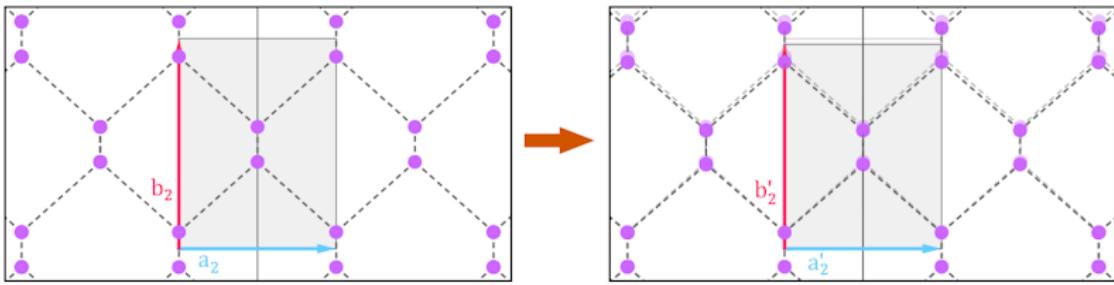


Figure 4.8: Deformation applied to the black phosphorene layer for the defined system. In this case, the deformation modifies PVs a and b such that: $\Delta M_a = 0.2\%$, $\Delta M_b = 2.8\%$, without affecting direction, so $\Delta A_a = \Delta A_b = 0^\circ$. This deformation results in a $DD = 0.009611$.

4.4 Calculation and Display of the First Brillouin Zone

As previously mentioned, one of the additional results provided by the program is an image of the first Brillouin zone (FBZ) of the primitive cells for each layer of the system, as well as an image of the FBZ for the entire system. These data can be helpful for users as they provide crucial information about the paths connecting high-symmetry points for determining electronic and phononic dispersion relations.

The FBZ of each 2D crystalline network corresponds to a parallelogram, depending on the symmetry of the Bravais lattice to which it belongs. To compute the FBZ, the Wigner-Seitz cell is calculated in reciprocal space using the reciprocal vectors of the network.

Thus, the FBZ corresponds to the convex polygon bounded by the perpendicular bisectors between the origin and the reciprocal lattice points. To define this polygon, we must determine the points at which its vertices lie, which occur where the bisectors intersect.

To calculate these points, the program utilizes a modified version of the *Simplex optimization algorithm*[23]² through the method `calcVerticesFBZ` in the `basics.py` file. The Simplex method can be adapted to traverse the perimeter of a convex figure in 2D because its algorithm systematically moves between the vertices of a convex polyhedron via its edges, optimizing an objective function at each step. In the case of a two-dimensional convex figure, such as the FBZ we seek, this principle is directly applicable, since the edges and vertices of the convex polygon represent the boundary of the feasible region. According to Bazaraa, Jarvis, and Sherali [24], the Simplex method employs a structured approach for traversing the edges of a polyhedron, allowing its strategy to be adapted for orderly traversal of a convex figure's vertices, ensuring continuous transit along its perimeter.

In this modified Simplex algorithm, operations are performed on a matrix formed by the equations of the computed perpendicular bisectors. The program uses the bisectors between the origin and the eight nearest reciprocal lattice points to determine the initial boundary, and from there, it traverses each polygon edge to find the vertices, stopping once the polygon is closed.

To illustrate the algorithm's operation, we will solve an example, showing geometrically what happens at each step.

Let L be a 2D oblique crystal lattice with primitive vectors $a = (2.5, 0.0)$ and $b = (-1.75, 3.0)$, for which we will calculate the FBZ. The first step is identifying the eight reciprocal lattice points closest to the origin, which in this case are:

$$\begin{array}{ll} V_1 = (0.0000, -2.0944) & V_2 = (0.0000, 2.0944) \\ V_3 = (-2.5133, 0.6283) & V_4 = (2.5133, -0.6283) \\ V_5 = (-2.5133, -1.4661) & V_6 = (2.5133, 1.4661) \\ V_7 = (-2.5133, 2.7227) & V_8 = (2.5133, -2.7227) \end{array}$$

Using these points v_i , we calculate the perpendicular bisector lines m_i between them and the origin (Figure 4.9).

Based on these lines, we create a table with 11 columns and 8 rows. The first 8 columns form an identity matrix of size 8×8 , while columns 9, 10, and 11 store the coefficients a , b , and c of the equations for each bisector m_i , with each row i corresponding to a bisector m_i , as shown in Table 4.1.

²Also known as the Nelder-Mead method or *Amoeba method*, among others.

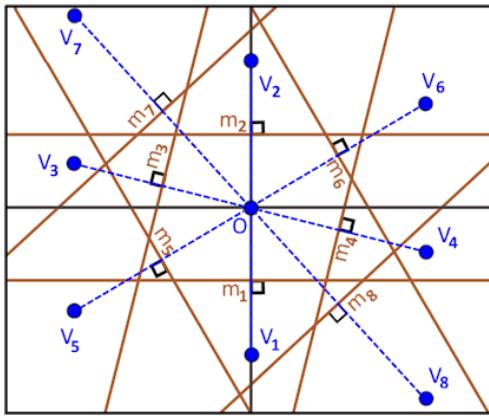


Figure 4.9: Closest reciprocal lattice points to the origin and their perpendicular bisectors.

$$\begin{array}{ll}
 m_1 : 0.00x - 4.19y = 4.39 & m_2 : 0.00x + 4.19y = 4.39 \\
 m_3 : -5.03x + 1.26y = 6.71 & m_4 : 5.03x - 1.26y = 6.71 \\
 m_5 : -5.03x - 2.93y = 8.47 & m_6 : 5.03x + 2.93y = 8.47 \\
 m_7 : -5.03x + 5.45y = 13.73 & m_8 : 5.03x - 5.45y = 13.73
 \end{array}$$

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-4.19	4.39
m_2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.19	4.39
m_3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	-5.03	1.26	6.71
m_4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	5.03	-1.26	6.71
m_5	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	-5.03	-2.93	8.47
m_6	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	5.03	2.93	8.47
m_7	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	-5.03	5.45	13.73
m_8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	5.03	-5.45	13.73

Table 4.1: Initial state of the table

Having located the pivot, we must *insert* row x . To do this, we perform operations so that, in the end, the column x has zeros in all its cells except in the pivot cell, where we must have a value of 1, arriving at a new table (Table 4.2).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-4.19	4.39
m_2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.19	4.39
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
x	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	1.0	-0.25	1.34
m_5	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	-4.19	15.18
m_6	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	4.19	1.75
m_7	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	4.19	20.44
m_8	0.0	0.0	0.0	-1.0	0.0	0.0	0.0	1.0	0.0	-4.19	7.02

Table 4.2: Table with a row corresponding to x .

The operations required to *replace* row a with a new row b in a matrix of real values, using the value in position $[a, b] = p$ as the *pivot*, are as follows:

1. Normalization of the pivot row:

- Divide all elements in row \mathbf{a} by the pivot value p , performing the operation $\mathbf{a} = \mathbf{a}/p$.
- Rename row \mathbf{a} as \mathbf{b} . This ensures row \mathbf{b} replaces the original row \mathbf{a} , and the value in the pivot position (now $[b, b]$) becomes 1.

2. Reduction of other rows:

- For each row \mathbf{c} different from row \mathbf{b} ($\mathbf{c} \neq \mathbf{b}$), perform the operation $\mathbf{c} = \mathbf{c} - q_c \mathbf{b}$, where q_c is the value in cell $[c, b]$ of row \mathbf{c} . This operation ensures that the value in position $[c, b]$ for every row \mathbf{c} (except for \mathbf{b}) becomes 0.

Following these steps, we obtain a new row \mathbf{b} in the position of row \mathbf{a} , ensuring that the entire column corresponding to \mathbf{b} contains zeros, except for position $[b, b]$, which holds the value 1 due to the normalization of row \mathbf{b} .

What we did by inserting row x is equivalent to casting a ray from the origin along the X axis, obtaining its intersections with the drawn perpendicular bisectors, and keeping the closest one to the origin³. In the example, this was the intersection of the X axis with line m_4 (Figure 4.10), positioning ourselves at a point on the FBZ perimeter of the network.

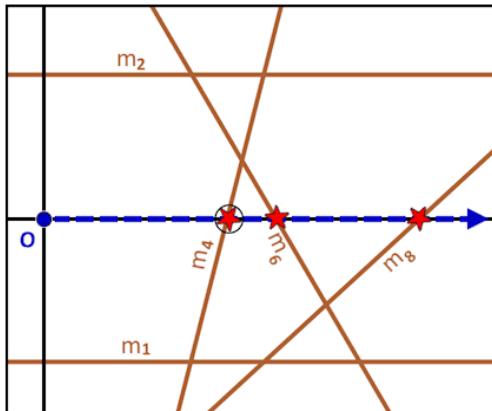


Figure 4.10: Visual representation of the first step of the algorithm, identifying intersections of the ray from the origin along the X axis with the calculated bisectors.

Now, to find the first vertex of the polygon, we must insert y into the rows. This step is similar to the previous one, but now we search for a pivot in column y instead

³This interpretation is directly derived from the geometric explanation of the Simplex method's initialization, where we initially position ourselves at a basic feasible solution—in this case, the highest point with $y = 0$ within the convex polygon bounded by the traced bisectors.

of column x . In this case, we choose cell $[m_6, y]$ as the pivot. Performing the required operations, we obtain a new table (Table 4.3).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
m_2	0.0	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	2.63
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
x	0.0	0.0	0.0	0.14	0.0	0.06	0.0	0.0	1.0	0.0	1.44
m_5	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
y	0.0	0.0	0.0	-0.24	0.0	0.24	0.0	0.0	0.0	1.0	0.42
m_7	0.0	0.0	0.0	2.0	0.0	-1.0	1.0	0.0	0.0	0.0	18.69
m_8	0.0	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	8.77

Table 4.3: Table after inserting both x and y . The first vertex of the polygon corresponds to the point $(1.44, 0.42)$.

Now, we attempt to insert row m_6 . In this case, the best pivot choice is $[x, m_6]$, but since this would remove x from the rows, we opt for the next best choice, $[m_3, m_6]$, and perform the corresponding operations to obtain a new table (Table 4.4).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.77
m_4	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_6	0.0	-1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	10.79
x	0.0	0.06	-0.2	0.0	0.0	0.0	0.0	0.0	1.0	0.0	-1.07
m_5	0.0	1.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	6.14
y	0.0	0.24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.05
m_7	0.0	-1.0	-1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	2.63
m_8	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	24.83

Table 4.4: Table after inserting m_6 and removing m_3 from the rows.

From this table, we obtain the point $P_3 = (-1.07, 1.05)$. Since this differs from P_1 , we continue (Figure 4.11).

Next, we attempt to insert row m_2 . The best pivot choice is $[y, m_2]$, but we discard this option and instead use the next best choice, $[m_5, m_2]$, generating a new table (Table 4.5).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	0.0	2.63
m_4	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_6	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
x	0.0	0.0	-0.14	0.0	-0.06	0.0	0.0	0.0	1.0	0.0	-1.44
m_2	0.0	1.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	6.14
y	0.0	0.0	0.24	0.0	-0.24	0.0	0.0	0.0	0.0	1.0	-0.42
m_7	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	8.77
m_8	0.0	0.0	2.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	18.69

Table 4.5: Table after inserting m_2 and removing m_5 from the rows.

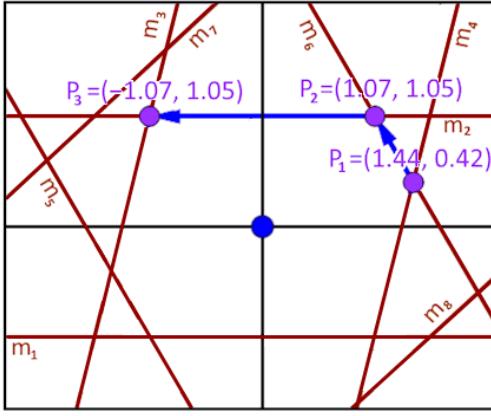


Figure 4.11: Third vertex found at the intersection of m_2 and m_3 .

From this table, we obtain the intersection between the lines m_3 and m_5 , $P_4 = (-1.44, -0.42)$ (Figure 4.12).

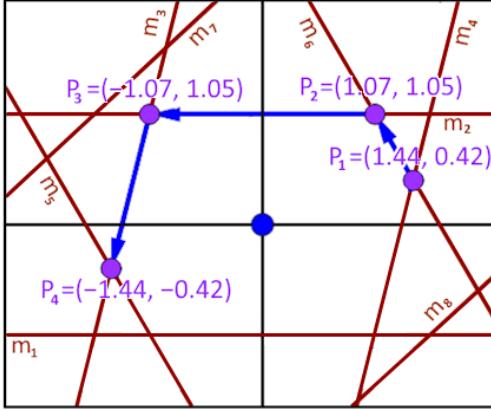


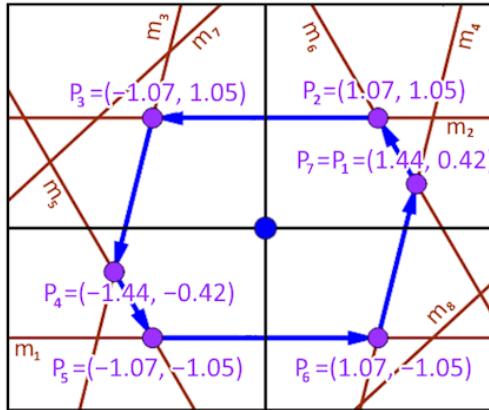
Figure 4.12: Fourth vertex found at the intersection of m_3 and m_5 .

We continue the algorithm a few more times, inserting m_3 and removing m_1 , then inserting m_5 and removing m_4 , obtaining the points $P_5 = (-1.07, -1.05)$ and $P_6 = (1.07, -1.05)$.

We now have Table 4.6, where the pivot is (m_1, m_6) . By removing m_6 to insert m_1 and performing the necessary operations, we obtain a new Table 4.7, which identifies a new edge at the point $P_7 = (1.44, 0.42)$. However, since this is the same as P_1 , we do not include P_7 as a new vertex and conclude the algorithm, determining that the FBZ of the network corresponds to the polygon $[P_1, P_2, P_3, P_4, P_5, P_6]$ (Figure 4.13).

Knowing the positions of the vertices defining the network's FBZ, we can represent it graphically in reciprocal space (Figure 4.14).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_5	-1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	10.79
m_6	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
x	-0.06	0.0	0.0	0.2	0.0	0.0	0.0	0.0	1.0	0.0	1.07
m_2	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.77
y	-0.24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	-1.05
m_7	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	24.83
m_8	-1.0	0.0	0.0	-1.0	0.0	0.0	0.0	1.0	0.0	0.0	2.63

Table 4.6: Table after several repetitions of the algorithm, obtaining points $P_1, P_2, P_3, P_4, P_5, P_6$.Figure 4.13: All vertices of the polygon forming the FBZ, tracing its search path from P_1 and ending at P_7 , which is the same point.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_4	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
m_6	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
x	0.0	0.0	0.0	0.14	0.0	0.06	0.0	0.0	1.0	0.0	1.44
m_2	0.0	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	2.63
y	0.0	0.0	0.0	-0.24	0.0	0.24	0.0	0.0	0.0	1.0	0.42
m_7	0.0	0.0	0.0	2.0	0.0	-1.0	1.0	0.0	0.0	0.0	18.69
m_8	0.0	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	8.77

Table 4.7: Final table state. Here, the values for cells (c, x) and (c, y) return to their original values from the start of the second phase, indicating the perimeter traversal is complete.

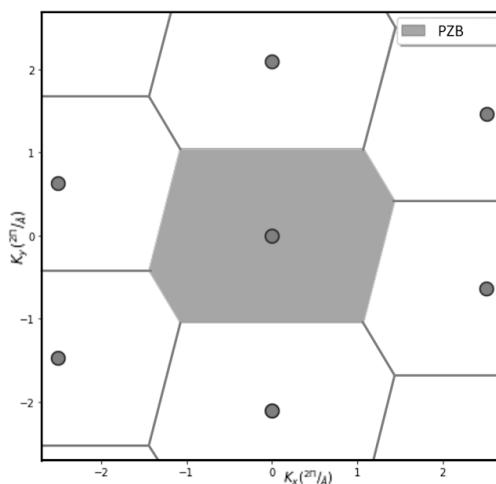


Figure 4.14: Image generated by the program of the evaluated network's FBZ, obtained from the described algorithm.

Functions

A.1 Basics

A.1.1 Vector Operations

Throughout the entire code, vector operations are performed. Since we are working with periodicity in two dimensions, most of the vectors used are two-dimensional. In this program, these vectors are represented as pairs of numbers indicating the position in the plane of the corresponding vector (their type would be described as `v2d::(float, float)`).

For dot product and cross product operations, we work with three-dimensional vectors. In these cases, a vector is represented by a set of three numbers that indicate the spatial position of the point represented by the vector (its type in this case would be described as `v3d::(float, float, float)`).

The functions that perform vector operations are listed below:

- **sumaV(\vec{a}, \vec{b})**:: `v2d, v2d → v2d`.
Adds vectors \vec{a} and \vec{b} .
- **multV(n, \vec{a})**:: `v2d, v2d → v2d`.
Multiplies vector \vec{a} by constant n .
- **m2V(\vec{a}, \vec{b}, s)**:: `v2d, v2d, (float, float) → v2d`.
Takes vectors \vec{a} , \vec{b} , and a pair $s = (m, n)$ and returns the linear combination $m\vec{a} + n\vec{b}$.
- **rota(\vec{v}, θ)**:: `v2d, float → v2d`.
Returns the result of rotating vector \vec{v} by angle θ degrees.
- **dist(\vec{a}, \vec{b})**:: `v2d, v2d → float`.
Computes the distance between the points represented by vectors \vec{a} and \vec{b} .
- **long(\vec{v})**:: `v2d → float`.
Computes the magnitude of vector \vec{v} .

- **cRot(\vec{v})**:: v2d → float.
Returns the direction of vector \vec{v} in degrees.
- **cAng(\vec{a}, \vec{b})**:: v2d, v2d → float.
Computes the angle between vectors \vec{a} and \vec{b} , returning a positive value (counter-clockwise) or negative (clockwise).
- **pC(\vec{a}, \vec{b})**:: v3d, v3d → v3d.
Computes the cross product of vectors \vec{a} and \vec{b} .
- **pP(\vec{a}, \vec{b})**:: v3d, v3d → v3d.
Computes the dot product of vectors \vec{a} and \vec{b} .
- **to2D(\vec{v})**:: v3d → v2d.
Returns the 2D projection of 3D vector \vec{v} .
- **to3D(\vec{v})**:: v2d → v3d.
Returns a 3D vector equivalent to 2D vector \vec{v} .

A.1.2 Matrix Operations

The program operates solely with 2×2 matrices, which are represented by a list of two lists, each containing two floats (`m2x2:: [[float, float], [float, float]]`).

Matrix operation functions include:

- **sumaM(A, B)**:: m2x2, m2x2 → m2x2.
Adds matrices A and B .
- **multM(c, M)**:: float, m2x2 → m2x2.
Multiplies matrix M by scalar c .
- **VtM(\vec{u}, \vec{v})**: Creates a 2x2 matrix from two 2D vectors used as columns.
- **MtV(M)**:: m2x2 → v2d, v2d.
Converts the two columns of matrix M into 2D vectors.
- **det(M)**:: m2x2 → float.
Computes the determinant of matrix M .
- **inv2x2(M)**:: m2x2 → m2x2.
Computes the inverse of matrix M .
- **m2M(A, B)**:: m2x2, m2x2 → m2x2.
Multiplies matrix A with matrix B .

- **transfVs(\vec{u}, \vec{v}, T)**:: v2d, v2d, m2x2 → v2d, v2d.
Returns two 2D vectors resulting from multiplying matrix $[\vec{u}, \vec{v}]$ by matrix T .

A.1.3 Auxiliary Functions

Below are the auxiliary functions defined in Basics.py used by subsequent routines.

- **getLim(\vec{u}, \vec{v}, m, n)**:: v2d, v2d, int, int → [float, float], [float, float].
Computes the min and max X and Y bounds of the rectangle enclosing the parallelogram spanned by $m\vec{u}$ and $n\vec{v}$.
- **esRotacion($\vec{a}, \vec{b}, \vec{c}, \vec{d}, \varepsilon$)**:: v2d, v2d, v2d, v2d, float → bool.
Determines whether the pair \vec{a}, \vec{b} are a rotation of \vec{c}, \vec{d} with an error tolerance ε (default $\varepsilon = 0.001$).
- **acomoda($x, valor, lis, tamMax$)**:: object, float, [[object, float]], float → [[object, float]].
Inserts object x into ordered list lis based on $valor$, limiting the list to $tamMax$ entries.
- **pmat(M)**:: m2x2 → .
Prints matrix M as a string.
- **checkP($x, y, err = 0.001$)**:: float, float, float → bool.
Checks if two values are equivalent within a periodic domain of size 1.
- **cRecip($\vec{a}, \vec{b}, \vec{c}$)**:: v3d, v3d, v3d → [v3d, v3d, v3d].
Given a triple of 3D vectors, returns the corresponding reciprocal vectors.
- **buscaEsquina($M, j, e1, e2$)**:: [[float]], int, int, int → int.
Auxiliary function used to calculate the vertices of the FBZ.
- **opera(M, e, s)**:: [[float]], int, int → .
Transforms matrix M so that column e contains all zeros except a 1 at row s .
- **dameVecinos(rv)**:: [v3d, v3d, v3d] → [(float, float)].
Returns the eight nearest lattice points to the origin given by the primitive vectors in rv .
- **califica($pos, lista$)**:: (float, float), [(float, float)] → bool.
Auxiliary to dameVecinos, determines if pos is not “covered” by any point in $lista$.
- **calcVerticesFBZ(rv)**:: [v3d, v3d, v3d] → [(float, float)], [[float]].
Calculates the vertices of the Wigner-Seitz cell for the lattice defined by rv .

- **aN(a)::str → float**

Returns the atomic number of element symbol a , or 0.5 if not found.

- **F_G(a,G)::(str,v2d) → float**

Computes the atomic form factor for element symbol a at reciprocal vector G^1 .

¹Data taken from International Tables for Crystallography: <http://it.iucr.org/Cb/ch6o1v0001/>

A.2 Atom

A.2.1 Initialization

The *Atom* object is initialized with the following parameters:

- **pos:: (float, float).**
Indicates the atom's relative position with respect to the primitive vectors \vec{a} and \vec{b} of the lattice.
- **posZ:: float.**
Indicates the atom's relative position with respect to the primitive vector \vec{c} of the lattice. If not specified, the default value is 0.0.
- **color:: str.**
Indicates the color used to draw the atom on screen. If not specified, the default is 'black'.
- **sig:: str.**
Identifier to specify the type of atom, recommended to use the atomic symbol. If not provided, the default value is 'C'.

These parameters generate the corresponding attributes of the Atom object.

A.2.2 Atom Methods

The methods for Atom are as follows:

- **__str__(): →str.**
Returns the atom as a string in the format: Atom.sig+str(Atom.pos).
- **setData(color, sig):: color, str→.**
Modifies the 'color' and 'sig' attributes of the Atom.
- **getPos(): →(float, float).**
Returns the 'pos' attribute.
- **clasifica(loa):: [[Atom]] →int.**
Given *loa*, a list of lists of Atoms classified by the 'sig' attribute. If a list in *loa* contains Atoms with the same 'sig' as the Atom, it adds the Atom to that list and returns 1. Otherwise, it creates a new list containing the Atom and returns 0.

A.3 Lattice

A.3.1 Initialization

Objects of class *Lattice* model crystal lattices. The input parameters to initialize this class are:

- **vA:: v2d.**
Primitive vector a of the crystal lattice.
- **vB:: v2d.**
Primitive vector b of the crystal lattice.
- **atm:: [Atom].**
Basis of the crystal lattice.
- **enls:: [((float,float),(float,float))].**
List of start and end positions of line segments representing atomic bonds. (Only required for on-screen representation. If not provided or an empty list is passed, the bonds will not be drawn, but this does not affect the POSCAR file generated when exporting the Lattice or System it belongs to).
- **name:: str.**
Name given to the crystal lattice.
- **detachment :: float.**
Indicates the thickness of the crystal lattice in Angstroms.
- **prof:: int.**
Indicates the layer this lattice belongs to if it is part of a stacked system. If not part of a system, its value is 1.

The class attributes generated from the input parameters are:

- **a:: v2d ; b:: v2d.**
Attributes representing the primitive vectors \vec{a} and \vec{b} of the lattice, directly taken from vA and vB.
- **OriginalA:: v2d ; OriginalB:: v2d.**
Attributes representing the rotated primitive vectors \vec{a} and \vec{b} such that vector \vec{a} is aligned to 0° .
- **prof:: int; detachment:: float; name:: str; enls:: [((float,float),(float,float))].**
Attributes that directly store the respective parameters.

- `theta:: float ; inAngle:: float.`

Angle attributes of the lattice: the first corresponds to the rotation required to reach the modeled lattice starting from the same lattice with \vec{a} at 0° . The second is the included angle between vectors \vec{a} and \vec{b} .

- `atms:: [[Atom]].`

List of lists of Atoms classified by the 'sig' attribute, derived from the corresponding parameter.

- `reciprocalVectors:: [v3d, v3d, v3d].`

Tuple of primitive vectors of the corresponding reciprocal lattice.

- `layerList :: [Lattice].`

If this Lattice models a crystal lattice formed by stacking two or more lattices, the corresponding Lattice objects are listed here.

A.3.2 Lattice Methods

- `__str__0:: →str.`

Expresses the Lattice as a string in POSCAR file format.

- `get_pv:: → v2d, v2d.`

Returns the primitive vectors \vec{a} and \vec{b} .

- `showme(x, y, x0, y0, t, iName, sampling, scalePosL)::`

`int, int, int, int, float, str, bool, bool →.`

Displays a representation of the superlattice given the values of x and y . The parameters $x0$ and $y0$ are used if the superlattice needs to be printed from a position other than the origin. The parameter t indicates the thickness used to draw atoms and bonds. The parameter $iName$ specifies the name of the PNG file in which the generated image will be saved (in the Images folder). If this parameter is not provided, the image will only be displayed on the screen. The boolean parameters `sampling` and `scalePosL` determine whether the primitive vectors \vec{a}, \vec{b} will be drawn in the image and whether the scale position will be placed on the left.

- `showPC(iName)::str→none`

Calls the `showme` method with the necessary parameters to print an image of only the primitive cell of the lattice, indicating its primitive vectors. Like the previous method, the parameter $iName$ is optional. If provided, the image will be saved under this name; otherwise, it will only be displayed on the screen without being saved.

- `addAtms(loa):: [Atom] →.`

Adds the atoms from list loa to the `atoms` parameter in an ordered manner.

- **showData()::→str**

Generates a POSCAR-formatted text containing the lattice data.

- **aligned()**

Rotates the lattice so that vector \vec{a} is aligned with the X-axis.

- **rotate(θ):: float →.**

Rotates the lattice by θ° .

- **alignedLattice():: → Lattice.**

Returns a copy of the lattice aligned with the X-axis.

- **mRot(θ):: $\theta \rightarrow$ Lattice.**

Returns a copy of the lattice rotated by θ° .

- **export(name):: str →.**

Exports the lattice in a POSCAR file <name>.vasp

- **setNewVectors($newA, newB$):: v2d, v2d →.**

Sets new values for the Lattice attributes a and b .

- **getVectors():: → v2d, v2d.**

Returns the Lattice attributes a and b .

- **getOV():: → v2d, v2d.**

Returns the Lattice attributes *OriginalA* and *OriginalB*.

- **nOAtms():: → int.**

Returns the number of atoms in the basis of the lattice.

- **loAtms():: → [[str, float, float, float, v2d]].**

This is an auxiliary function used to calculate the diffraction pattern of the lattice. Generates a list with data from each Atom object in the lattice basis (*atms* attribute of the *Lattice* class). Each element in the output list contains the following data extracted from each atom: attribute *sig*, relative X-position, relative Y-position, relative Z-position, and a 2D vector with the absolute position of the atom's projection onto the plane.

- **F_hkl(h, k, FG)::float, float, bool → float**

This is an auxiliary function used to calculate the diffraction pattern of the lattice. Computes the *structure factor* corresponding to the reciprocal lattice point h, k, l (since the structure is 2D, only h and k are used).

- **reciprocalBackgroundMesh($vl, t, border, calcS, rnd, FG$)::**

$[(\text{float}, \text{float})], \text{float}, \text{float}, \text{bool}, \text{int}, \text{bool} \rightarrow$

[float], [float], [float], linkList.

This is an auxiliary function used to calculate the diffraction pattern and reciprocal space representation of the lattice. Computes the necessary data to draw the reciprocal space of the lattice. The list *vl* contains the vertices of the FBZ of the lattice, the displayed space is limited by *border*, *rnd* specifies the maximum number of digits after the decimal point when rounding values in the *S* list (included in the output of this function). The boolean variables *calcS* and *FG* determine whether to compute the structure factor (stored in list *S*) and whether the dispersion vector should be considered in the calculation.

The output of this method consists of three lists of real values and a vertex list. The first two lists (*X*, *Y*) store the positions of reciprocal lattice points within the space determined by *border*, the third list (*S*) contains the computed structure factor values for each point (empty if *calcS* = *False*), and the vertex list holds the vertices required to construct a mesh while calculating the FBZ at each point.

- **printReciprocalSpace(*t, border, prnt, zoom, colors*)::**

float, float, bool, bool, [str] →.

Displays a representation of the reciprocal space of the lattice on the screen. If the Lattice corresponds to a stacked network structure, it draws the FBZ of each constituent network and the overall system. The values *t* and *border* determine the general thickness used for drawing and the drawing space limit, respectively. If *prnt* = *True*, the image is saved in the “image” folder; if *zoom* = *True*, the displayed image corresponds only to the first quadrant of the plane. The list *colors* specifies the colors used to draw the FBZ of each lattice layer in the system; if provided, it must contain a color for each layer; otherwise, colors are assigned randomly.

- **printLightPoints(*t, border, prnt, rnd*)::** float, float, bool, int → none

Displays a representation of the diffraction pattern of the lattice. The values *t* and *border* determine the general thickness used for drawing and the drawing space limit, respectively. The parameter *rnd* specifies the number of decimal places used when rounding the structure factor at each point. If *prnt* = *True*, the image is saved in the “image” folder.

A.4 Functions

A.4.1 Functions on Atoms in an Atomic Basis

- **isitin**($r, cent, sr, slvl$):: Lattice, v2d, Lattice, int →.
Verifies which atoms belonging to the cell of the network r located at position $cent$ fall within the area corresponding to the PC of network sr . The atoms that satisfy this condition are added to the atomic basis of sr .
- **megeCut**(r, sr, lvl):: Lattice, Lattice, int →.
Identifies which cells of network r intersect the PC of sr and applies the *isitin* function on them.
- **cleanA**(r, err) :: Lattice, float →.
Verifies the atoms belonging to the atomic basis of network r and removes duplicates. The variable err indicates an acceptable error threshold; if not provided, the default value is 0.001.
- **esClon**($Atms, atm, \varepsilon$):: [Atom], Atom, float → bool
Determines if atom atm is equivalent to any element in the list $Atms$. This auxiliary function is used to remove duplicate atoms in the atomic basis of a network.
- **borders**($Atms$):: [Atom] → [Atom]
Given a list of atoms corresponding to the atomic basis of a network, returns a sublist containing those that are near the edge of the PC of the network.
- **cleanA**($Atms, \varepsilon$):: [Atom], float → [Atom]
Evaluates an atom list, identifying those equivalent by periodicity within an error threshold ε . Atoms identified as duplicates of an existing one are removed from the original list and stored in a new list returned by this function.
- **cleanPCell**(L, acc):: Lattice, int → [Atom]
Executes the *cleanA* function on the atomic basis of network L with $\varepsilon = 1/10^{\{acc\}}$.

A.4.2 Primitive Cell Calculation for a Network

Some of these functions use a list with a specific format for evaluating primitive vectors (PRs): $pList::[[([(int, int)], v2d)], float]]$. This is a list of paired elements where the first part indicates the PR index and its associated vector, while the second part specifies the computed *error* for the PR.

- **pts**(l, max_it):: Lattice, int → pList.
Creates a list with the PRs of l contained within the supercell defined by max_it . The result is returned in a list pts formatted as $pList$.

- **calcCD(s, l, ab)**:: Lattice,Lattice,(int,int) → (int,int).
Computes the indices c, d associated with the PR of network l closest to the PR of s referenced by index a, b . Uses Equation 4.2.
- **calcPR($pts, substrate, l, \varepsilon$)**:: pList,Lattice,Lattice,float → pList.
Filters the list pts , eliminating entries whose PR in l exceeds the given ε threshold. If ε is not provided, the default value is 0.05.
- **commonVs(Ls, max_val, ε)**:: [Lattice],int,float → [(int,int),float]
Using the network at position 0 in the list Ls as the ‘substrate’, sequentially executes the function calcPR for each additional network in Ls , with the initial pts list resulting from running ‘ $pts(substrate, max_val)$ ’. The final list includes only PRs with a corresponding PR in each system layer within the given ε error.
- **corresponding_points($l1, l2, M1$)**:: Lattice,Lattice,m2x2 → m2x2
Returns the 2×2 matrix referencing the PR for $l2$ corresponding to the PR of $l1$ referenced by $M1$.
- **calc_dd(Mi, Mo)**:: m2x2,m2x2 → float
Computes the *deformation index* for Mi if deformed into Mo .

A.4.3 Transformation Matrix Data Handling

The displayed MT data is extracted from a formatted list Mdata.

Mdata:: [m2x2,m2x2,float,float,float].

In order, the first matrix corresponds to the MT whose information is shown, the second is the associated deformation matrix, while the final four values indicate the deformation effect on the primitive vectors of the network—the first two for magnitude change, and the last two for directional change.

- **textLonN(t, n, al)**:: str,int,str → str
Returns a string with length n , either truncated or padded with spaces. The variable al determines text alignment when padding spaces: $al = l$ adds spaces to the right, $al = r$ adds spaces to the left, and $al = c$ distributes spaces evenly on both sides. If unspecified, the default is $al = c$.
- **header()**:: → str
Generates the header string used in tables displaying MT data.
- **infLayer($L, data$)**:: Lattice,Mdata → str,int
Generates a string displaying the information contained in $data$ using network L . Returns the generated string along with the number of atoms in the supercell of L corresponding to the given MT data.

A.4.4 Exporting a Network from a POSCAR File

- **readFile(name)::str→[str]**
Reads the file named *name* (relative to the code location) and treats it as plain text, converting its content into a list with each line as an element.
- **leeNumeros(s)::str→[float]**
 Parses text string *s* and returns a list containing all numeric values extracted from it.
- **importLattice(name, prnt)::str, bool→Lattice**
Creates a Lattice object from the POSCAR-format file *name*. If *prnt* = *True*, a message displays indicating the creation of the Lattice object and the filename from which it was generated.

A.4.5 Predefined Lattices

These functions generate predefined Lattice objects for hexagonal and rectangular networks, as well as some common crystalline structures.

- **hexa6(p, atms, name)::float, [Atom], str→Lattice**
Generates a hexagonal lattice with 6 radial symmetries, using lattice constant *p*, atomic basis *atms*, and name *name*. The *atms* list is optional; if omitted, the lattice will contain 2 atoms in its basis.
- **graphen()::**
Generates a graphene lattice with lattice constant 2.44Å and centered atoms forming a hexagonal lattice with 6 radial symmetries.
- **blackPhosphorene()::**
Generates a black phosphorene lattice with lattice constants 3.2601Å and 4.347Å.

A.5 System

A.5.1 Initialization

Objects of the *System* class model structures (systems) formed by vertical stacking of two-dimensional crystalline networks. The **input parameters** for initializing this class are:

- **lol::[Lattice]**

List of stacked crystalline networks that constitute the system. The order of the list determines the stacking sequence of the networks, such that the network at position i is immediately above the network at position $i - 1$. The network at position 0 corresponds to the bottom layer, called the substrate layer.

- **name::str**

Name used to identify the system.

The **Attributes** of this class are:

- **redes::[Lattice]**

List of stacked crystalline networks that constitute the system.

- **name::str**

Name of the system.

- **poits::[[int,int],float]**

List containing all PRs of the substrate layer's network, linked to the VTs common to all networks in the system. When initialized, this attribute is an empty list.

- **loMat::[m2x2]**

List containing the MTs that indicate a possible PC for the system. When initialized, this attribute is an empty list.

- **MaxNumM::[int]**

Indicates the maximum number of matrices stored in the *loMat* list. When initialized, this attribute is set to 10.

- **SuperRed::Lattice**

Stores the Lattice that represents the complete system based on the computed PC. When initialized, this attribute is set to "None".

- **MT::m2x2**

Stores the MT that defines the computed PC. When initialized, this attribute is set to "None".

A.5.2 System Methods

Main Methods

- **searchLP(*rangeOfSearch, epsilon*)**:: int, float → int

Computes and stores in the system's *points* attribute a list of all PRs common to all networks in the system within the search area specified by *rangeOfSearch* with an error lower than *epsilon*.

- **calculateTM(*min_angle*)**:: bool → int

This method computes the MTs that define a PC for the system based on the VTs specified in the system's *poits* list. The computed MTs are stored in the *loMat* list. If the *poits* list is empty, an error message is displayed and -1 is returned. If it identifies that all networks in the system are hexagonal, it computes only MTs that generate hexagonal PCs and returns 0. If **not** all networks in the system are hexagonal, it returns 1. The variable *min_angle* specifies the minimum internal angle that the PCs corresponding to the MTs must have; the default value is 20° .

- **createSuperLattice(*M*)**:: m2x2 → int

Creates and stores in the system's *SuperRed* attribute the network representing the complete system using the PC defined by *M*. If this network is successfully created, 1 is returned; otherwise, 0 is returned. The network generated by this method may contain imperfections as it **does not** deform the networks that make up the system.

- **optimize_system(*T, prnt*)**:: m2x2, bool → System, [m2x2]

Creates a copy of the system and deforms its networks such that, once these deformed networks are transformed, they do not generate imperfections when periodically replicating the PC resulting from the *createSuperLattice* method on the deformed copy with the MT *T*. The *SuperRed* and *MT* attributes from the copy are transferred to the original system, and the modified system along with a list of deformation matrices used in each layer is returned.

- **ShowTMs(*shw, save*)**:: bool, str → m2x2

Performs an analysis of all MTs in *loMat*, identifying and returning the MT with the lowest distortion degree (DD)². If *shw* = *True*, it displays tables with the characteristics of the PCs linked to the MTs in *loMat*. Providing the variable *save* is optional; if given, a text file named *save* is created containing the evaluated tables.

- **manualAdjustment(*TMs*)**:: [m2x2] → [m2x2]

Deforms the networks in the system so that, when transformed by the MTs described in *TMs*, the resulting supercells have the same shape and dimensions. The required deformation matrices are returned in a list.

²This measure is calculated using Equation 4.10 in the Deformation Matrix Calculation section of the Implementation chapter.

Quick Methods

These methods sequentially execute some main methods to streamline instructions.

- **ejecuta(n, e)**: int, float → m2x2

Sequentially executes the functions *searchLP*, *calculateTM*, and *ShowTMs* without displaying messages on the screen. Returns the MT associated with the smallest-distortion PC for the search area specified by n with an error lower than e .

- **generateSuperCell(RoS, ϵ_s, sd)**: int, float, bool → System

Sequentially executes the functions *searchLP*, *calculateTM*, *ShowTMs*, and *optimize_system*, returning a deformed system with its calculated PC based on the MT obtained from the *ShowTMs* function, which corresponds to the MT associated with the smallest-distortion PC for the search area specified by *RoS* with an error lower than ϵ_s .

Parte II

Versión en Español

Introducción

En la naturaleza existen sistemas a escala atómica¹ cuya razón entre superficie y volumen es $\geq 10^3$ lo que permite clasificarlos como materiales bidimensionales (2D). Uno de los materiales 2D más conocidos y estudiados desde su síntesis por exfoliación mecánica en 2004 por Geim y Novoselov es el grafeno [1]. El grafeno es un sistema compuesto de átomos de carbono dispuestos en un arreglo hexagonal en forma de panal de abeja con grosor de un átomo. El grafeno se puede encontrar como una lámina atómica apilada verticalmente formando grafito, el cual es aquel que encontramos en la punta de un lápiz, figura 1.1(a). Vale la pena mencionar que las fuerzas que mantienen unidas a las láminas de grafeno es mucho más débil que aquellas fuerzas que mantienen unidos a los átomos en el plano.²

Como consecuencia del trabajo de Geim y Novoselov, surgió el interés por el estudio teórico y/o experimental (o la combinación de ambos) de otros materiales 2D; experimentado un aumento notable del interés en la comunidad científica. Debido a que sistemas como el grafito están compuestos por láminas de grafeno débilmente unidas, es posible pensar que otros sistemas al igual que el grafeno como el disulfuro de molibdeno (MoS_2) sean utilizados para construir, usando un mecanismo de apilamiento vertical, un sistema heterogéneo de dos o más láminas con propiedades físicas distintas, creando así lo que se denomina heteroestructuras. Estas heteroestructuras cuentan con potenciales aplicaciones en la industria electrónica y optoelectrónica [2, 3].

La formación de estas heteroestructuras es posible debido, nuevamente, a las fuerzas de interacción entre las capas atómicas, que son significativamente más débiles que las fuerzas que mantienen unidos a los átomos dentro de cada capa. Para ser preciso, el tipo de fuerzas débiles son de carácter dispersivo de tipo van der Waals (vdW).³ A este tipo de sistemas se les conoce como homo- y hetero-estructuras de van der Waals. El apilamiento de las capas se puede vislumbrar cómo algo semejante a un juego de LEGO,

¹En escala de 10^{-10} metros.

²Las fuerzas que mantienen unidas a las láminas son de carácter dispersivo, mientras que aquellas que unen a los átomos son por enlaces químicos.

³Interacciones dipolo-dipolo son las más representativas.

donde las láminas atómicas se ensamblan como bloques de construcción capa por capa, ver figura 1.1(b).

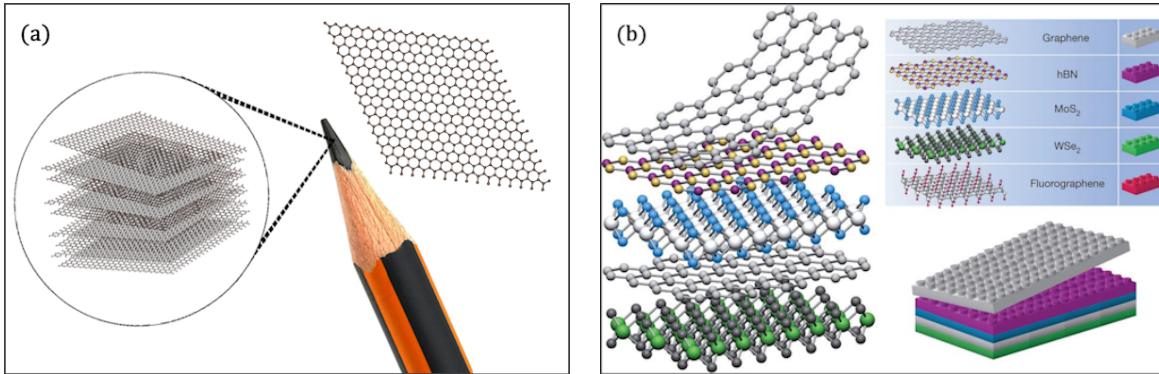


Figura 1.1: (a) Grafeno extraído del grafito usado en lápices. (b)Panel derecho: Láminas atómicas que pueden ser apiladas verticalmente donde cada esfera de diferente color representa una especie química diferente. Panel izquierdo: Analogía entre sistemas laminares atómicos con bloques de construcción de Lego y su correspondiente apilamiento vertical. Figura tomada del artículo “*Van der Waals heterostructures*”[4].

Mediante el apilamiento vertical de materiales 2D, es posible crear nuevas estructuras que, en relación con las propiedades físicas de los materiales individuales, pueden conservar dichas propiedades, mejorarlas o incluso exhibir características completamente nuevas no observadas en los materiales aislados. Además, los sistemas formados por capas de los mismos materiales pueden mostrar propiedades diferentes si presentan variaciones como el orden de apilamiento o distintas rotaciones en sus capas. Todo esto dota al apilamiento vertical de materiales 2D de un gran potencial para crear materiales con diversas propiedades físicas, lo que hace que estas estructuras generadas sean de gran interés para la Física y las Nanociencias, con posibles aplicaciones en Nanotecnología [5].

Ejemplos del uso de las heteroestructuras de van der Walls en la actualidad se observan en:

- La fabricación de semiconductores, lo que ha favorecido en la miniaturización de circuitos integrados, sustituyendo las conexiones alámbricas de berilio por capas de aluminio y mejorando su funcionalidad al emplear elementos compuestos por diferentes tipos de semiconductores [6, 7].
- Experimentos donde se busca limpiar aguas residuales mediante fotocatálisis utilizando su banda energética [8].
- La creación de fotodetectores de alta sensibilidad, capaces de operar en un amplio rango spectral, desde el visible hasta el infrarrojo [3].

- La creación de láseres ultradelgados y flexibles muy prometedores para aplicaciones en pantallas, comunicaciones ópticas y tecnologías de realidad aumentada [9].
- La fabricación de LEDs y celdas solares ultradelgadas aplicables en dispositivos portátiles y en tecnología flexible, una tendencia creciente en la industria [10].

Como se mencionó antes, el grosor de los materiales 2D utilizados está en el orden de angstroms (\AA)⁴ y sus interacciones físicas ocurren a escala atómica, por lo que el estudio de estos sistemas recae en el área de la Física del Estado Sólido. Esta área de la física tiene como objetivo principal el estudio de las propiedades físicas de estructuras cristalinas, basándose en el hecho de que dichas estructuras presentan una periodicidad en el espacio real, ya sea en una (1D), dos (2D) o tres dimensiones (3D)[11] (figura 1.2). Basados en su periodicidad, estas estructuras cristalinas, conocidas simplemente como **cristales**, pueden describirse mediante una **celda primitiva**, que se compone de dos elementos fundamentales: *una red de Bravais* y *una base atómica*.

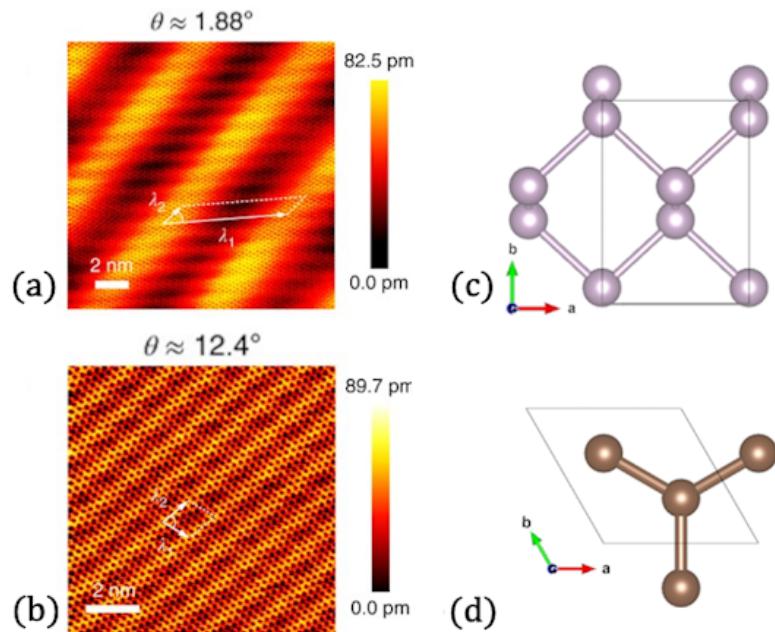


Figura 1.2: Imágenes tomadas con microscopio de efecto túnel (STM por sus siglas en inglés) de heteroestructuras formadas por dos capas de fosforeno negro y una de grafeno con una rotación de (a) $1,88^\circ$ y (b) $12,4^\circ$ donde se puede apreciar la periodicidad que presentan estas [12]. Representación gráfica de las celdas unitarias del (c)fosforeno negro y (d)grafeno.

La red de Bravais representa la repetición translacional de los átomos en el espacio real mientras que la base atómica indica la disposición específica de los átomos dentro de la

⁴ $1\text{\AA} = 0,1\text{ nm} = 1 \cdot 10^{-10}\text{ m}$

celda (figura 1.3). La naturaleza cuántica de estos sistemas y sus características a escala atómica hacen que la determinación y comprensión de sus celdas primitivas sean de vital importancia para el estudio de sus propiedades físicas y químicas como son la estructura electrónica y reactividad química.

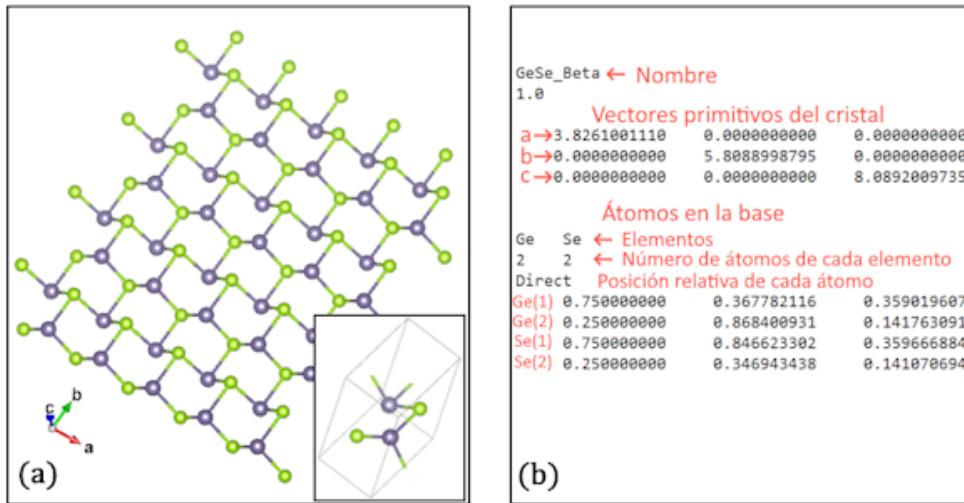


Figura 1.3: (a) Lámina de Germanio-Selenio fase Beta (β -GeSe) y su celda primitiva. (b) Imagen de un archivo en formato POSCAR donde se observan los vectores que generan la Red de Bravais y la base atómica del β -GeSe

En la actualidad, para el estudio teórico de estos sistemas, estructuras compuestas de dos o más materiales 2D, se utilizan herramientas computacionales como la *Teoría de los Funcionales de la Densidad* (DFT, por sus siglas en inglés)[13]. En estos casos es necesario identificar y caracterizar el “*bloque de construcción*” más pequeño del sistema, la celda primitiva. Una vez hecho lo anterior si se pretende estudiar al sistema perturbado es necesario construir otro sistema periódico más grande, a partir de la celda primitiva, que es conocido en el contexto de Estructura Electrónica y Estado Sólido como *supercelda*.

La obtención de estas superceldas no es un problema trivial a diferencia de lo que hace parecer la visualización de estos sistemas con piezas de LEGO apiladas en la figura 1.1(b), dado que la celda primitiva de cada una de las capas del sistema puede tener diferente forma, tamaño y orientación. Así, determinar una supercelda (común) que tenga la misma periodicidad en todas las láminas atómicas no es sencillo y su tamaño llega a ser muy grande, con varios miles de átomos en esta. Es importante mencionar que, en ciertas condiciones, la existencia de una supercelda común en un sistema puede no lograrse si no se aplica en este un efecto de tensión sobre las capas que lo conforman, obteniendo así un nuevo sistema aproximado al original, el cual sí presenta una supercelda que mantenga la periodicidad de todas las capas en todo el espacio real como en la figura 1.4.

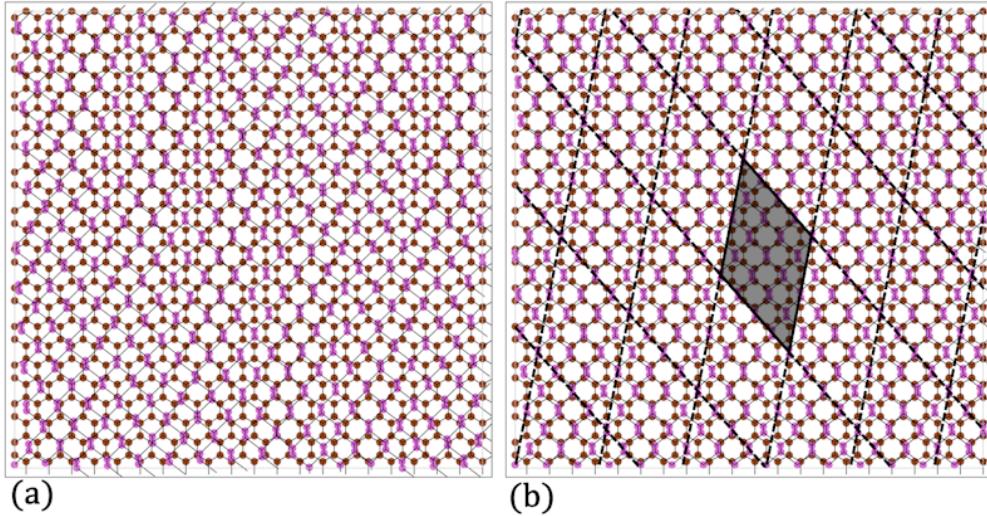


Figura 1.4: Sistema bicapa de grafeno-fosforeno negro con un ángulo relativo entre capas de $3,5^{\circ}$. En (a) se muestra el sistema sin alteración en las capas y no se encuentra una supercelda que pueda mantener la periodicidad completa dentro del área observada, sin embargo en (b) se muestra el sistema optimizado, donde la capa de fosforeno negro sufrió una deformación aumentando el tamaño de sus vectores primitivos en un $-1,94\%$ y $+1,90\%$ respectivamente, en este sí existe una supercelda que mantiene la periodicidad en todo el sistema.

Por todo lo anterior, la obtención de una celda mínima para una red cristalina que pueda describir estructuras formadas por dos o más capas de materiales 2D, es el primer gran problema a resolver antes de poder estudiar las propiedades físicas que puedan presentarnos dichas estructuras. Existen implementaciones publicadas actualmente que atacan este problema, tres de estas son los códigos *CellMatch* [14], *Supercell-core software* [15] y *Twister* [16].

Sin embargo, pese a que el problema se ha pensado solucionar con estas herramientas, aún nos topamos con algunas complicaciones, por un lado, en la implementación *CellMatch* se hace uso de fuerza bruta para encontrar su respuesta, lo que afecta su eficiencia computacional, además de que esta implementación solo trabaja con estructuras bicapa⁵.

Por otra parte, en la implementación de *Twister*, el principal problema a resolver es la obtención de ángulos de rotación óptimos, exclusivamente para sistemas de 2 capas, limitando el tipo de sistemas que se pueden resolver con este y dejando en segundo plano

⁵Al iniciar este trabajo la página señalada por el artículo, esta era un “enlace roto”, por lo que no era posible acceder a su código, impidiendo su uso o revisión. Sin embargo en una revisión posterior el enlace ya era válido, permitiendo su análisis.

la obtención de la supercelda para el sistema, así que para esto, dicho código utiliza un algoritmo de “*fuerza bruta*”, que conlleva a tener una solución con complejidad en $O(n^4)$, teniendo como n el límite en el espacio de búsqueda de la supercelda.

Por último, en *Supercell-Core* se tiene un programa más completo y robusto que los dos anteriores; sin embargo, mientras que sus resultados en sistemas específicos son buenos, en otros sistemas (sistemas más complejos que presentan superceldas muy grandes) aún muestra problemas, teniendo resultados con átomos faltantes o extras en su base atómica, esto posiblemente por problemas en el redondeo a la hora de efectuar las operaciones o por los métodos de deformación empleados, estos problemas no son fácilmente detectables en sistemas muy grandes lo que puede llevar a errores.

Por lo tanto, el objetivo general de este proyecto como una Tesis de Licenciatura, es desarrollar un código computacional abierto (*open source*)[17], en lenguaje Python[18], que sea flexible, robusto e intuitivo para el usuario. Este código tiene como fundamentos el uso de métodos geométricos[19, 20] para la determinación de una supercelda 2D (pequeña) commensurable para sistemas compuestos por dos o más láminas atómicas con diferentes orientaciones relativas y quiralidad, asegurando que presente una tensión acotada en cada capa.

La información resultante del código incluirá los vectores de la red en su representación matricial, así como las posiciones de los átomos en coordenadas relativas a los vectores de la red, con la posibilidad de exportar dichos resultado a un archivo POSCAR⁶. Esta información puede ser utilizada en estudios de primeros principios para caracterizar la estructura electrónica del sistema o para posibles optimizaciones estructurales mediante el uso de herramientas de dinámica molecular con recursos de supercómputo.

Con este código, además de abordar las limitaciones observadas en trabajos previos, también se proporcionará información sobre el espacio recíproco correspondiente a los sistemas calculados, información que no ha sido considerada en los resultados arrojados por ninguna de las implementaciones mencionadas anteriormente.

Conocer la imagen de un cristal en el espacio recíproco es fundamental en la física del estado sólido y en la teoría de estructura electrónica de materiales a escala atómica. El espacio recíproco es crucial para comprender la dispersión de las ondas y los fenómenos de difracción, como los observados en la difracción de rayos X y la microscopía electrónica de transmisión de alta resolución (HRTEM, por sus siglas en inglés). Estos fenómenos permiten obtener información detallada sobre la estructura cristalina y la disposición de

⁶Un archivo POSCAR es un formato estándar para describir la estructura cristalina de un material en términos de los vectores de red y las posiciones de los átomos en una celda unitaria.

los átomos en un material.

Además, la imagen en el espacio recíproco proporciona información sobre la periodicidad y la simetría de la red cristalina, información esencial para comprender las propiedades físicas y electrónicas de los materiales. Por ejemplo, los picos en un patrón de difracción de rayos X o en una imagen de HRTEM corresponden a vectores de red recíproca que están relacionados con la estructura cristalina del material. Estos picos pueden proporcionar información sobre la estructura cristalina, la orientación de los cristales y las deformaciones en la red.

En resumen, esta información adicional que será proporcionada por el programa es esencial para caracterizar la estructura y las propiedades de los materiales a nivel atómico, lo que tiene implicaciones significativas en áreas como la ciencia de materiales, la nanotecnología y la electrónica.

El enfoque interdisciplinario de esta Tesis combina conceptos de física de materiales, cristalográfica, programación y simulación computacional, lo que permitirá abordar sistemas complejos y utilizar métodos avanzados de análisis. Además, que el código sea abierto es especialmente importante, ya que permitirá que otros investigadores lo utilicen y contribuyan al desarrollo tanto del software como de sus propios proyectos científicos.

Estructura general del programa

El programa que aquí se presenta, está formado por 6 archivos de python, *Basics.py*, *Atom.py*, *Lattice.py*, *Functions.py*, *System.py* e *Interface.py*. Estos archivos son importados cada uno por el siguiente archivo, haciendo una especie de encapsulado (figura 2.1). De tal manera que las funciones descritas en cada uno de ellos son ocupadas por los siguientes archivos, manteniendo un orden y jerarquía, permitiendo al usuario interactuar solamente con la capa más externa, *Interface.py*, donde se guía al usuario paso a paso desde la consola para crear y analizar un sistema¹. A continuación se describirá, brevemente, cada uno de los archivos antes señalados².

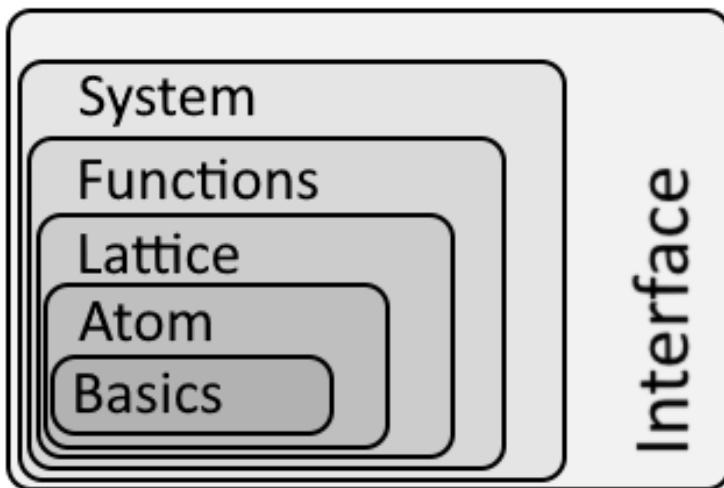


Figura 2.1: Representación esquemática de la estructura general del programa. Esta figura muestra como cada nivel encapsula al anterior hasta llegar a la interface del usuario.

¹En caso de que el usuario quiera usar las herramientas del programa de forma directa y manual basta con importar el archivo *System.py* para tenerlas todas esas herramientas a disposición

²Todas las funciones pertenecientes a cada archivo se tratan con profundidad en el apéndice B.

2.1 Basics.py

Este archivo agrupa funciones esenciales que constituyen los cimientos del programa. Las funciones pertenecientes a este bloque pueden clasificarse en tres grupos: funciones que operen sobre vectores, funciones que operan sobre matrices y funciones misceláneas. Estas últimas son funciones usadas por algunos métodos que resuelven problemas específicos. Ejemplos de estas funciones son las utilizadas para encontrar los puntos de red de vecinos más cercanos a uno específico; o la función que calcula los vértices de una Celda de Wigner-Seitz.

El propósito de escribir manualmente estas funciones básicas aquí es para tener una independencia de bibliotecas externas a la biblioteca estándar de Python, proporcionando portabilidad al programa y eliminando la necesidad de que los usuarios instalen bibliotecas de Python de terceros. Además de asegurar un control más detallado de las operaciones internas del programa.

2.2 Atom.py

En este archivo se describe la clase “Atom”, clase que sirve de molde para crear los objetos ocupados en este programa para modelar los átomos que conforman la base atómica de los cristales.

Esta clase es muy básica y maneja solamente la información mínima de los átomos en un cristal requerida por el programa, lo que, aunque pasa por alto diversas características en el modelado, es suficiente para los requerimientos del programa.

La robustez de la clase “Atom” radica en su capacidad para representar y clasificar átomos pese a su simplicidad, proporcionando una estructura coherente y organizada para construir estructuras más complejas.

2.3 Lattice.py

Esta clase es usada para crear objetos que se erigen como la representación integral de los cristales en este programa. Son definidos por sus vectores primitivos de red y la base atómica proporcionada por una lista de objetos “Atom”.

Además de la inicialización del objeto “Lattice”, esta clase alberga funciones que operan directamente sobre él. Estas funciones desempeñan un papel crucial en la mani-

pulación y análisis de los cristales.

En esta clase se describen desde funciones para modificar el objeto mismo, hasta funciones con el propósito de exportar la información del cristal hacia un archivo POSCAR o desplegar una representación del cristal en pantalla.

2.4 Functions.py

El archivo functions.py contiene diversas funciones que trabajan mediante la interacción de objetos tipo “Lattice” principalmente. La principal función de las funciones en este archivo es la de encontrar superceldas de dimensiones similares comunes en todos los objetos de esta lista y conocer el ajuste que estos objetos requieren para lograr la coincidencia sin ser deformados más allá de un valor límite específico.

Además de las funciones anteriores, en este archivo también se tienen funciones que crean de forma rápida objetos tipo “Lattice” hexagonales y rectangulares, donde solo se dan como parámetros de entrada el tamaño de el o los periodos (en Å) de la red.

También se cuenta con funciones que inicializan “Lattices” específicas para algunos materiales recurrentes como por ejemplo: *grafeno*, *fosforeno negro*, y las fases α y β del compuesto GeSe, por simplicidad α -GeSe y β -GeSe.

2.5 System.py

En este archivo se describe la clase “System”, en esta se crean objetos los cuales actúan como el modelo para las heteroestructuras con las que se está trabajando. La base fundamental de estos objetos es una lista con las “Lattices” correspondientes a los materiales que apilados verticalmente conforman a la heteroestructura por modelar.

Las funciones de esta clase se encargan de operar sobre el objeto System, encontrando posibles celdas primitivas para este, analizando los resultados obtenidos o se encargan de presentar estos resultados al usuario de forma que el usuario haga uso de lo que más le convenga dado el problema que quiere resolver.

2.6 Interface.py

Este archivo viene siendo la capa más externa del programa. Es la que interactúa directamente con el usuario dándole acceso a la creación y manipulación de los objetos "System", para modelar las estructuras cuya celda primitiva se desea conocer.

Algoritmo principal

En el análisis del software presentado, se modelará a la estructura bajo estudio como S , un objeto de clase “*System*” definido por una lista de objetos de tipo “*Lattice*”.¹ Estos objetos son la representación de las redes periódicas 2D asociadas a las estructuras que son apiladas para formar la heteroestructura. Una vez obtenida la celda primitiva para S , se define un nuevo objeto *Lattice* llamado **SuperRed** asociado a S ($S.SuperRed$), de tal forma que S pueda ser visto como una red periódica 2D.

Internamente, el programa identificará al primer elemento de la lista de redes que definen a S ($S.Layers[0]$) como la **capa sustrato**, esta es la capa sobre la cual se realizan los principales análisis dado que esta capa no tendrá deformaciones al momento de resolver el problema.

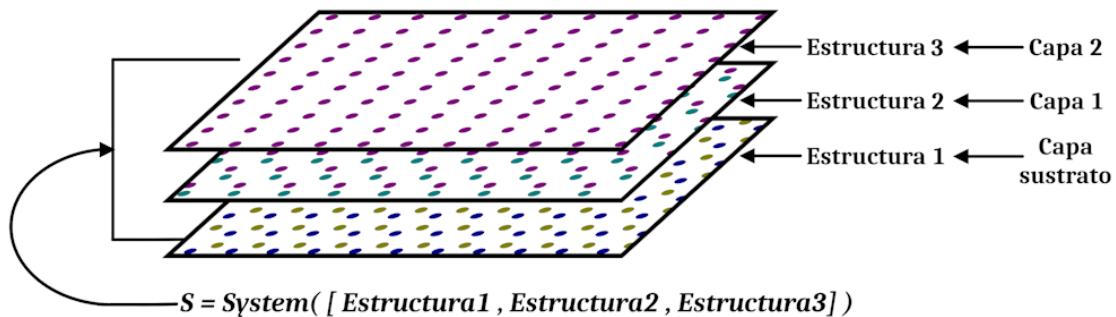


Figura 3.1: Representación general sobre cómo el software abstraе el sistema a partir de la lista de estructuras iniciales.

Fue mencionado anteriormente que, si encontramos superceldas para las redes en cada una de las capas de un sistema que coinciden al proyectarlas sobre el plano, entonces tenemos una posible celda primitiva para el sistema (ver figura 3.2). De manera inversa, una celda que cumple con ser una celda primitiva para el sistema S coincidirá con alguna supercelda de cada una de las estructuras que conforman dicho sistema. Además, los vectores que describen estas superceldas serán vectores de traslación válidos para estas

¹Ambas clases, *System* y *Lattice*, se verán a detalle más adelante en este capítulo.

redes.

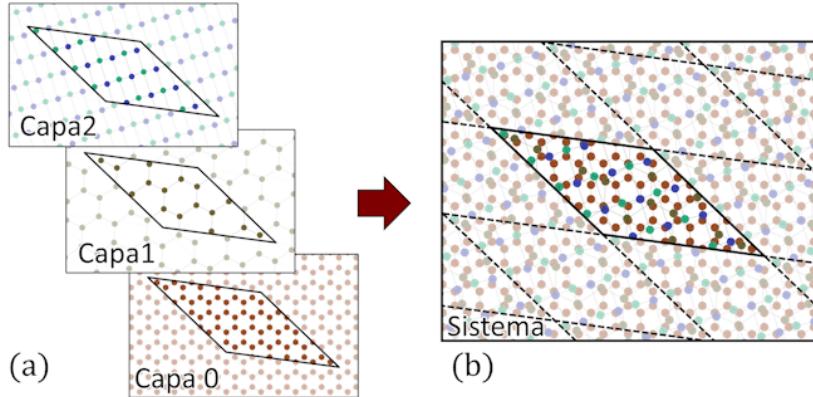


Figura 3.2: La figura en (a) muestra superceldas de 3 redes cristalinas distintas pertenecientes a las capas de un sistema apilado, estas superceldas tienen la cualidad de ser idénticas al proyectarse sobre el plano. En la figura (b) se presentan una celda primitiva para la heteroestructura, esta celda coincide con el apilamiento de las superceldas en (a).

Esta propiedad muestra que los vectores primitivos del sistema coinciden con algún vector de traslación de las redes en cada capa. Por lo tanto, es posible relacionar las celdas primitivas del sistema con una matriz de transformación (MT) para alguna capa del sistema. En el programa esta identificación se hará con la correspondiente MT de la capa sustrato del sistema, esto ya que, como se mencionó, esa capa no será modificada durante todo el proceso.

En estructuras 2D, una MT es una matriz 2×2 formada por los coeficientes enteros de los vectores de traslación que describen la supercelda ligada a la MT (un ejemplo se ve en la figura 3.3). Esta MT se construye de la siguiente forma: Sean \mathbf{a} y \mathbf{b} los vectores primitivos de una red cristalina 2D, L , entonces la MT correspondiente a la supercelda formada por los vectores de traslación $\mathbf{s}_a = (m\mathbf{a} + n\mathbf{b})$ y $\mathbf{s}_b = (p\mathbf{a} + q\mathbf{b})$ es

$$T = \begin{pmatrix} m & p \\ n & q \end{pmatrix}$$

Dado lo anterior, lo primero a hacer para resolver el problema principal es encontrar los VTs comunes en todas las capas, que serán posibles vectores primitivos del sistema. Para llevar a cabo la búsqueda de estos vectores, el programa necesita parámetros adicionales. En primer lugar, es necesario delimitar el espacio de búsqueda, lo cual se logra mediante un valor entero n , este valor define un espacio de búsqueda, el cual corresponderá al paralelogramo formado por la supercelda de la capa sustrato de S , desde $-n$

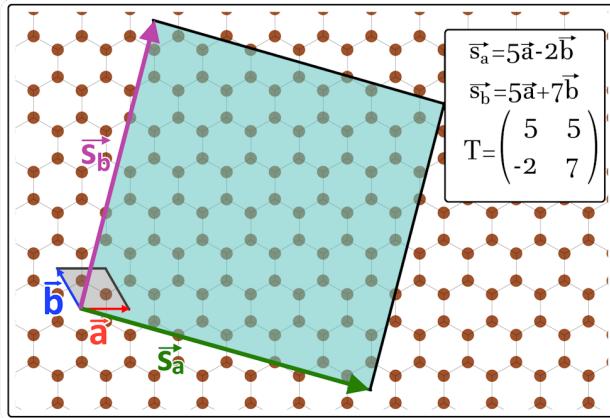


Figura 3.3: La MT, T , hace referencia a la supercelda cuadrada definida por los vectores de traslación \mathbf{s}_a y \mathbf{s}_b , en una red hexagonal que tiene como vectores primitivos, \mathbf{a} y \mathbf{b} .

hasta n , en ambos vectores primitivos \mathbf{a} y \mathbf{b} como se representa en la figura 3.4.

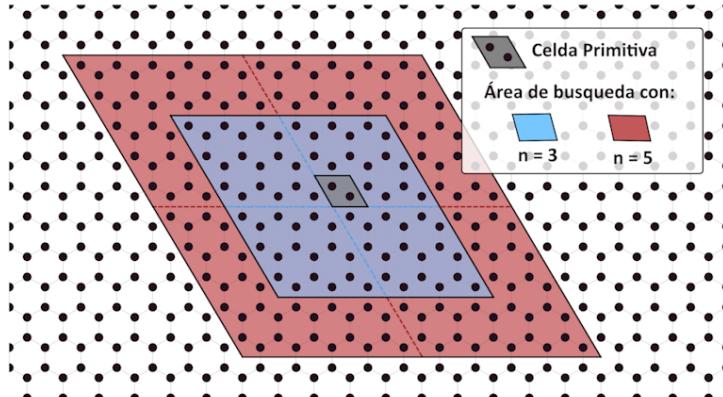


Figura 3.4: En rojo y azul las áreas de búsqueda delimitadas por $n = [-3, 3]$ y $n = [-5, 5]$ para una red hexagonal. Se puede observar que el área de búsqueda crece de forma cuadrática con respecto al valor de n que la genera.

El segundo parámetro a considerar es una constante ε que representa un límite máximo en el error permitido entre los vectores de traslación similares en todas las capas. Un menor valor de ε permite resultados que requieren una menor deformación mecánica (compresión o estiramiento) de las redes para lograr la periodicidad, sin embargo puede requerir una mayor área de búsqueda para tener resultados.

En el algoritmo, los posibles vectores primitivos para S se definen mediante una lista de puntos de red (PRs), R , esta lista se obtiene a partir de una lista de los PRs de la capa sustrato ubicados dentro del área de búsqueda especificada. Esta lista es “limpiada” dejando solo aquellos PRs que presenten un error menor o igual al ε dado con los PRs

correspondientes a cada capa del sistema. Obteniéndose así una lista con solo los PRs que se repiten en todas las capas del sistema con un error acotado.²

Posteriormente, a partir de la lista R , se generan las posibles MTs que darán forma a la supercelda de la capa sustrato a partir de la cual se definirá la celda primitiva para S . Estas matrices se construyen seleccionando parejas de vectores de traslación de R , asegurándose que estos vectores generen MT válidas.

Para que una matriz sea considerada una MT válida para una red debe de cumplir que los vectores resultantes de esta sean linealmente independientes, para lo que basta con que su determinante sea diferente a cero.

De las matrices calculadas, se crea la lista *LoMat* de S , *S.LoMat*, en la cual se almacenan aquellas MTs correspondientes a las superceldas más pequeñas. En esta etapa, el usuario tiene la opción de decidir manualmente una MT, T , de *LoMat* o permitir que el programa utilice la que ha calculado como la mejor opción.

Para poder elegir la MT más conveniente para el usuario se puede hacer uso del método *ShowTMs* de la clase *System*, la cual despliega en pantalla tablas con columnas que muestran las características que tendría la celda primitiva generada para cada MT en *LoMat*. Estos datos son:

- **Lattice:** Nombre de la red perteneciente a la capa.
- **T:** MT que construye la supercelda correspondiente a la celda primitiva en dicha capa.
- **Deformation:** Matriz de deformación que requiere la red para mantener la periodicidad.
- **Distortion $\delta // \theta$:** Cambio sufrido por los vectores primitivos de la red tras la deformación, donde δ es el cambio en su magnitud y θ el cambio en su dirección.
- **#Atoms:** Número de átomos contenidos en cada capa.

Para finalizar se muestra el número total de átomos en la celda primitiva asociada a cada MT y una calificación *DD* (Degree of distortion); la cual está relacionada al porcentaje de distorsión total que debe sufrir el sistema para ser periódico con esta celda primitiva³.

²El algoritmo empleado aquí será descrito con detalle más adelante.

³La matriz de deformación, los cambios δ y θ y la DD se describirán más adelante en este capítulo.

***Option 1: T <- Matrix loMat[0]					
Lattice	T	Deformation	Distortion: δ/θ	#Atoms	
Grafeno	1 3 -7 -2	1.00000 0.00000 0.00000 1.00000	+0.0% // +0.0° +0.0% // +0.0°	38	
Grafeno(13.15°)	-1 2 -8 -3	1.00024 -0.00047 0.00047 0.99976	+0.0% // +0.02° +0.0% // +0.02°	38	
Black-Phosphorene(27.50°)	1 2 -4 -2	1.07381 0.04535 0.12880 1.07834	+4.129% // -1.53° -2.132% // +2.22°	24	

Total atoms:100 DD: 2.5147

***Option 2: T <- Matrix loMat[1]					
Lattice	T	Deformation	Distortion: δ/θ	#Atoms	
Grafeno	3 9 -2 1	1.00000 0.00000 0.00000 1.00000	+0.0% // +0.0° +0.0% // +0.0°	42	
Grafeno(13.15°)	2 8 -3 -1	0.97872 -0.01482 -0.03541 0.97584	+0.0% // +0.02° +0.0% // -1.51°	44	
Black-Phosphorene(27.50°)	2 6 -2 -2	0.97132 -0.05713 0.03181 0.98134	-2.132% // +2.22° -2.609% // +2.28°	32	

Total atoms:118 DD: 2.6946

Figura 3.5: Tablas correspondientes a MTs, *LoMat[0]* y *loMat[1]*, calculadas para una heteroestructura formada por dos capas de grafeno y una de fosforeno negro. La capa de grafeno superior tiene una rotación con respecto a la inferior de 13.15° , y la capa de Fosforeno está rotada 27.5° con respecto a la misma capa de grafeno. En este caso, la MT recomendada es *LoMat[0]*, esto es por su menor distorsión y tamaño, el cual es de solo 100 átomos.

Para finalizar, teniendo una MT adecuada T , se calcula la celda primitiva para S usando el método *S.createSuperLattice(T)*, el cual crea un nuevo objeto tipo *Lattice* para S llamado *Supercelda*. Esta corresponde a la red formada por la celda primitiva generada por T . El resultado puede ser presentado en pantalla con una proyección 2D en el espacio real de la celda primitiva calculada y una representación en el espacio recíproco del sistema S con el método *show*, ver figura 3.6.

Como acción extra, si se requiere, mediante la función *exportLattice* de la clase *System* se puede crear un archivo POSCAR⁴ para la supercelda calculada, este archivo puede ocuparse como entrada para cálculos *ab initio* [21], o para ser visualizado con paquetes de cristalográfica [22].

Al analizar la complejidad del algoritmo principal, observamos que está determinada principalmente por los algoritmos utilizados para generar la lista de vectores R y la lista de matrices *LoMat*. La complejidad de estos algoritmos depende directamente del númer-

⁴Este formato es utilizado como archivo de entrada, con información de la geometría y composición del sistema, en códigos computacionales de primeros principios que emplean la DFT como VASP (Vienna *ab initio* Simulation Package).

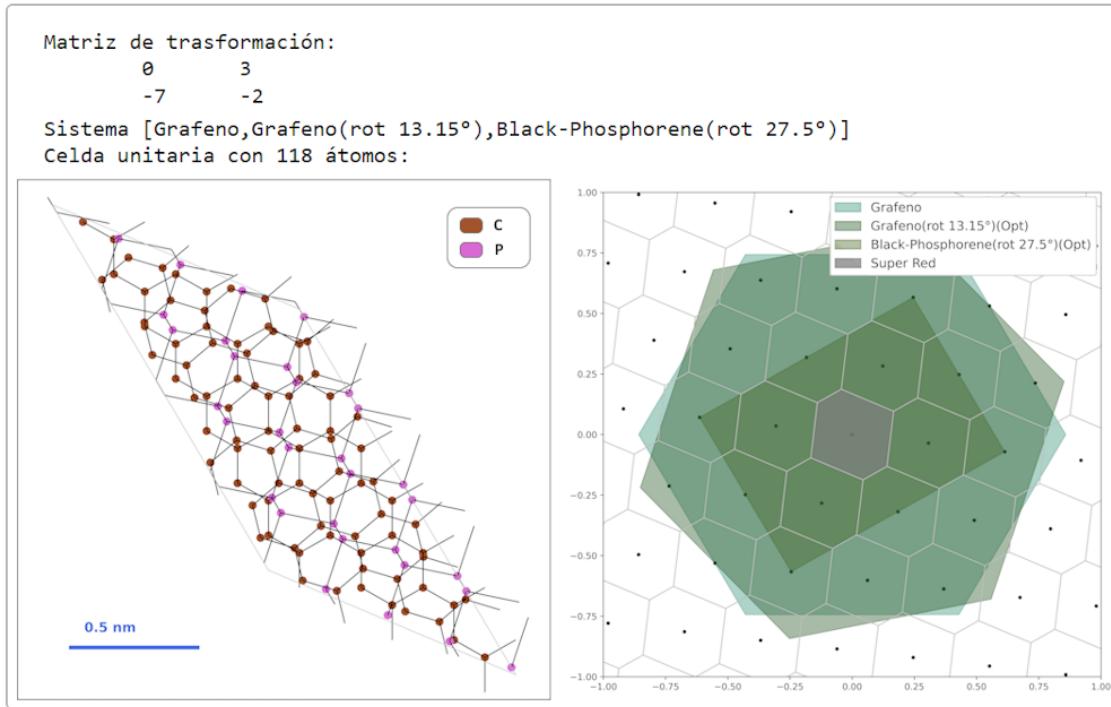


Figura 3.6: Imágenes desplegadas en pantalla al generar la supercelda del sistema compuesto por dos capas de grafeno y una de fósforo negro. El panel izquierdo muestra una representación 2D en el espacio real de la supercelda calculada. El panel derecho muestra la correspondiente representación en el espacio recíproco. Los puntos de la red, en el espacio recíproco son identificados por puntos negros que son encerrados por celdas hexagonales deformadas como aquella en morado centrada en el origen. Por otra parte, en este mismo espacio recíproco de la supercelda, es posible dibujar también las primeras zonas de Brillouin de cada celda primitiva: Dos hexagonales para grafeno, y una rectangular para fosforeno.

ro de puntos de red en la capa de sustrato dentro del espacio de búsqueda designado, con una complejidad de $O(n^2)$ para la generación de R y $O(n^4)$ para la generación de $LoMat$ ⁵. En consecuencia, la complejidad general del algoritmo está limitada por la creación de la lista $LoMat$, lo que implica un orden de complejidad de $O(n^4)$. Sin embargo, como se analizará más adelante, en la práctica la complejidad tiende a ser considerablemente inferior a este límite teórico.

⁵Estos resultados se verán en el análisis de dichos algoritmos en la siguiente sección de este mismo capítulo.

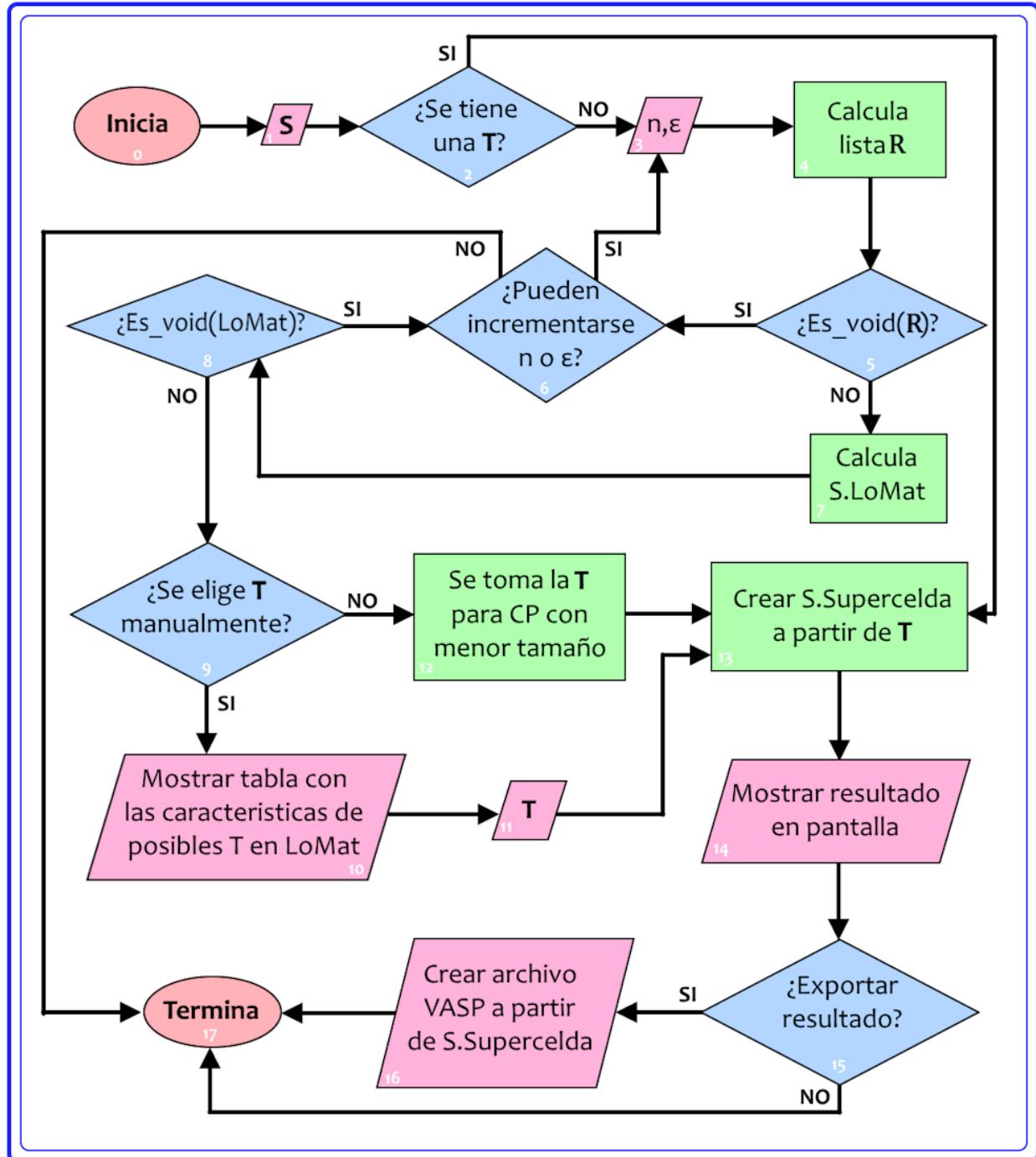


Figura 3.7: Diagrama de flujo del algoritmo general que sigue el programa para encontrar la supercelda.

El algoritmo principal del programa puede resumirse de la siguiente manera:

Algoritmo principal	(3.1)
<ol style="list-style-type: none"> 0. Inicia. 1. Crea S. 2. Si se cuenta con una matriz de transformación T se salta al paso 11, de lo contrario continúa. 3. Se define n y ϵ para la búsqueda. 4. Calcula la lista R con los vectores de traslación que satisfacen el límite de error ϵ para todas las capas de S. 5. Si R está vacía continúa, de lo contrario salta al paso 7. 6. Si se pueden dar nuevos valores mayores para n o ϵ se actualizan y se regresa al paso 3. De lo contrario no se puede encontrar una celda primitiva para S en el área de búsqueda y se termina el algoritmo (salto a 17). 7. Calcula una lista <i>LoMat</i> de S. 8. Si <i>LoMat</i> está vacía se regresa al paso 6, de lo contrario se continua. 9. Si se desea elegir una T manualmente a partir de las matrices en <i>LoMat</i> se continua, caso contrario se salta al paso 12. 10. Se despliega en pantalla una tabla con los datos de las matrices de transformación calculadas en <i>LoMat</i>. 11. Ingresa la T y salta al paso 13. 12. Escoge la mejor T de las MT de <i>LoMat</i>. 13. Calcula la Lattice que funcionará como CP para S a partir de la matriz T obtenida. 14. Muestra en pantalla el resultado. 15. Si se desea exportar la supercelda calculada se continua, de lo contrario el algoritmo finaliza. 16. Se crea un archivo POSCAR con los datos de la supercelda calculada. 17. Finaliza el algoritmo. 	

Métodos y algoritmos esenciales

En esta sección, se abordan los algoritmos centrales usados en la solución del problema, así como los métodos principales que los implementan. Estos algoritmos son aquellos que por su complejidad e importancia sobresalen entre todos los demás.

La explicación detallada de estos conjuntos de instrucciones proporcionará una comprensión profunda de su funcionalidad y la teoría subyacente que los sustenta. Este análisis tiene como objetivo no solo demostrar la robustez de dichos métodos, sino también arrojar luz sobre su aplicación práctica y su relevancia en la solución del problema para la investigación en materiales avanzados.

4.1 Búsqueda de candidatos a vectores primitivos

Siguiendo el algoritmo principal (algoritmo 3.1), si desconocemos una MT T que resuelva el problema, entonces, una vez determinados los valores n y ε , lo primero a hacer es calcular una lista de posibles vectores primitivos, R , para el sistema evaluado, S , estos vectores deben estar en el área de búsqueda delimitada por n y tener un desajuste menor que ε .

Para llevar a cabo esta tarea, se requieren dos cosas. La primera es un método que nos permita encontrar el VT \mathbf{u}_L en una red L , que sea el más parecido al VT \mathbf{u} en otra red. La segunda es una forma de medir la diferencia, desajuste o error entre el vector original y el aproximado. Con estas dos herramientas, podremos obtener la lista deseada.

4.1.1 Determinar \mathbf{u}_L para un \mathbf{u} .

El encargado de este trabajo en el programa es el método llamado `calcCD()`, localizado en el archivo *Functions.py*.

Para iniciar, en el problema se tienen 2 redes, S y L . En la red S el vector \mathbf{u} es un vector de traslación válido y se requiere encontrar el vector \mathbf{u}_L , el cual cumple con ser el vector de traslación válido en L más parecido al vector \mathbf{u} . Para resolver este problema, lo primero es darle uso al concepto de puntos de red¹. De esta manera se puede simplificar una red bidimensional a un conjunto de puntos en el plano colocados de forma periódica, lo que elimina mucho ruido y simplifica el problema.

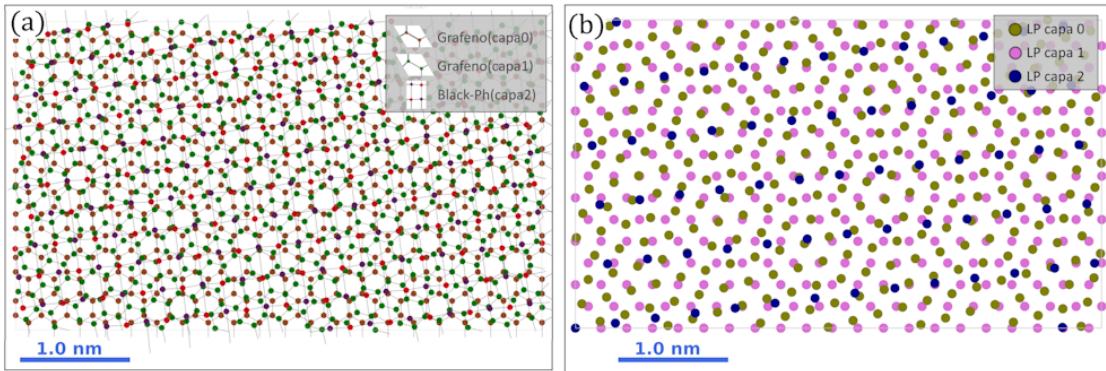


Figura 4.1: (a) Proyección en el plano de un sistema formado por dos capas de grafeno y una de fosforeno negro. La rotación entre la capa 0 y 1 es de $15,5^{\circ}$, mientras que la rotación entre la 0 y 2 es de $7,3^{\circ}$. (b) Representación en el plano de los PRs de las tres capas del sistema compuesto.

Sean los VPs de una red 2D \mathbf{a} y \mathbf{b} , con un punto fijo en el plano \mathbf{P}_0 , como un PR origen; entonces tomando en cuenta que, por periodicidad, todo PR corresponde a la traslación del punto \mathbf{P}_0 con un VT T y que:

$$T = i\mathbf{a} + j\mathbf{b}$$

entonces se puede relacionar cada PR (y al VT que la define), con un par de valores enteros.

$$\mathbf{P}_{i,j} = \mathbf{P}_0 + \mathbf{T}_{i,j} = \mathbf{P}_0 + i\mathbf{a} + j\mathbf{b} \quad \text{con } i, j \in \mathbb{Z} \quad (4.1)$$

Entonces, el sistema puede verse como un grupo de familias de puntos colocados de forma periódica en el plano (una por cada capa del sistema), donde todas las familias

¹Puntos discretos en el espacio que definen la periodicidad de una red cristalina. Estos puntos corresponden a las posiciones donde las unidades de la estructura del cristal (como átomos, iones o moléculas) se repiten de manera idéntica, según un conjunto de vectores de traslación definidos por los parámetros de la celda unitaria.

contienen al punto \mathbf{P}_0 . Así el problema de encontrar un VT común entre 2 redes de un sistema, se traduce a encontrar un punto diferente a \mathbf{P}_0 compartido en sus familias correspondientes.

Dado que los VP de una red U en 2D deben ser linealmente independientes, estos pueden ser usados como una base β (en este caso para \mathbb{R}^2). Entonces puede observarse que si se usa β como una base para el plano, en cada posición $(x, y)_{[\beta]}$, con $x, y \in \mathbb{Z}$, se obtiene el PR, $\mathbf{P}_{x,y}$, de la red U . Explotando el fenómeno anterior se planea lo siguiente. Se define la notación $\mathbf{T}_{R,i,j}$ como el vector de traslación de la red R cuyas componentes son los enteros i y j , además sean $\mathbf{P}_{i,j}$ los PRs correspondientes a S y $\mathbf{Q}_{k,l}$ los PRs correspondientes a L .

Si se tiene un vector $\mathbf{u} = \mathbf{T}_{S,a,b}$ es posible ubicar su correspondiente PR $\mathbf{P}_{a,b}$ en el plano. Suponiendo que $\mathbf{Q}_{c,d}$ es el PR de L más cercano a $\mathbf{P}_{a,b}$, entonces al hacer un cambio de base usando β (la base formada por los VP de L), se consigue una nueva posición para $\mathbf{P}_{a,b}$, aquí $\mathbf{P}_{a,b}$ seguirá siendo cercano a $\mathbf{Q}_{c,d}$ (dado que este cambio de base no es siempre una transformación *isométrica* esto no se puede generalizar, sin embargo, en vecindades muy pequeñas se puede suponer que se cumple).

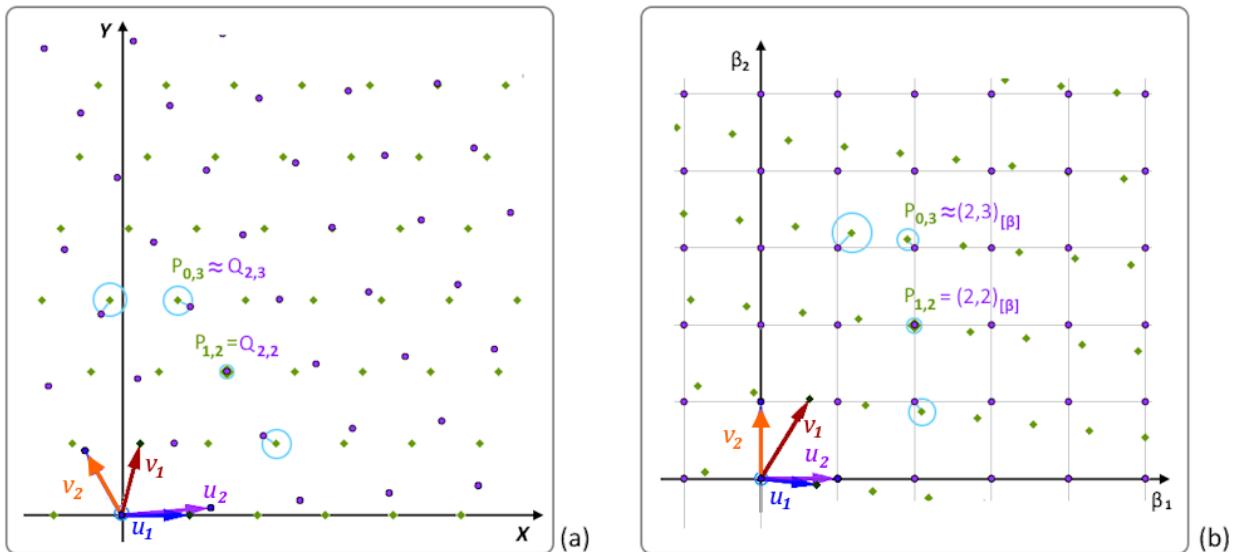


Figura 4.2: (a) Proyecciones en el plano de los PRs para 2 redes, una con VP $\mathbf{u}_1, \mathbf{v}_1$ (rombos verdes) y otra con VP $\mathbf{u}_2, \mathbf{v}_2$ (círculos morados). Se muestra cómo, si en (a) la base canónica para el plano, un PR $\mathbf{P}_{a,b}$ es muy cercano a un PR $\mathbf{Q}_{c,d}$, entonces en (b) el plano con base $\beta = \{\mathbf{u}_2, \mathbf{v}_2\}$, se tiene que $\mathbf{P}_{a,b} \approx (c, d)$.

Entonces, como se dijo antes, la posición de $\mathbf{P}_{a,b}$ será cercana a la posición de $\mathbf{Q}_{c,d}$ en la base β ; pero $\mathbf{Q}_{c,d}$, bajo la nueva base, está posicionado en unas coordenadas enteras. Para ser más preciso, en las coordenadas (c, d) . Por lo que la nueva posición de $\mathbf{P}_{a,b} \approx (c, d)_{[\beta]}$,

como se ejemplifica en la figura 4.2. Así, la forma en que encontramos estos enteros c, d es obtener la posición de $P_{a,b}$ en base β , $(P'_x, P'_y)_{[\beta]}$, y redondear al par entero más próximo, teniendo $\text{round}((P'_x)_{[\beta]}) = c$ y $\text{round}((P'_y)_{[\beta]}) = d$.

De forma general, sean $[\mathbf{u} = (u_1, u_2), \mathbf{v} = (v_1, v_2)]$ y $[\mathbf{w} = (w_1, w_2), \mathbf{z} = (z_1, z_2)]$ los VPs de las redes S y L , respectivamente y sea $T_{L,c,d}$ el VT de L más parecido a $T_{S,a,b}$; tenemos que:

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix} = \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \quad y \quad \begin{pmatrix} w_1 & z_1 \\ w_2 & z_2 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \end{pmatrix}_{[\beta]} = \begin{pmatrix} P_x \\ P_y \end{pmatrix}$$

Entonces:

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix}_{[\beta]} = \begin{pmatrix} w_1 & z_1 \\ w_2 & z_2 \end{pmatrix}^{-1} \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

Por lo tanto:

$$\begin{pmatrix} P'_x \\ P'_y \end{pmatrix}_{[\beta]} = \begin{pmatrix} \frac{u_1z_2 - u_2z_1}{w_1z_2 - z_1w_2} & \frac{v_1z_2 - v_2z_1}{w_1z_2 - z_1w_2} \\ \frac{u_2w_1 - u_1w_2}{w_1z_2 - z_1w_2} & \frac{v_2w_1 - v_1w_2}{w_1z_2 - z_1w_2} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \frac{a(u_1z_2 - u_2z_1) + b(v_1z_2 - v_2z_1)}{w_1z_2 - z_1w_2} \\ \frac{a(u_2w_1 - u_1w_2) + b(v_2w_1 - v_1w_2)}{w_1z_2 - z_1w_2} \end{pmatrix}$$

Así se tiene que:

$$\begin{aligned} c &= \text{round}((a(u_1z_2 - u_2z_1) + b(v_1z_2 - v_2z_1)) / (w_1z_2 - z_1w_2)) \\ d &= \text{round}((a(u_2w_1 - u_1w_2) + b(v_2w_1 - v_1w_2)) / (w_1z_2 - z_1w_2)) \end{aligned} \tag{4.2}$$

Con las Ecs. (4.2) podemos definir la función “*calculaCD*”, la cual al recibir 2 redes S, L y los coeficientes enteros a, b de $T_{S,a,b}$, regresará los coeficientes enteros c, d de $T_{L,c,d}$.

4.1.2 Cálculo del error para los vectores de translación

Esta función desempeña un papel fundamental en la funcionalidad total del programa, ya que es usada como el primer filtro para llegar a un resultado final. El propósito principal radica en establecer un criterio de evaluación para cada vector candidato a VP, con el fin de discernir cuáles son las mejores opciones entre ellos.

Para poder elegir entre los candidatos a VP, es necesario definir una función, $\text{err}(\mathbf{u}, \mathbf{u}_L)$ que evalúe el error $\epsilon_{\mathbf{u}_L}$ de cada VT \mathbf{u} en la capa sustrato, con respecto a un vector \mathbf{u}_L , el cual corresponde al VT de la red L que es más parecido a \mathbf{u} . Esta función debe cumplir lo siguiente:

- El valor de $\varepsilon_{\mathbf{u}_L}$ debe ser proporcional a la deformación requerida en la red en la capa L para mantener la periodicidad en toda la estructura.
- Para cada \mathbf{u} debe existir un único valor $\varepsilon_{\mathbf{u}_L}$ que lo califique.
- El valor $\varepsilon_{\mathbf{u}_L}$ debe ser siempre no negativo.
- Si $\varepsilon_{\mathbf{u}_L} = 0$, entonces el resultado es óptimo. Mientras mayor sea su valor, menos elegible es el vector \mathbf{u} como VP.

Que el valor de $\varepsilon_{\mathbf{u}_L}$ sea no negativo y que una calificación cercana a cero sea el resultado óptimo, permite una fácil interpretación de la calificación de \mathbf{u} . Además de que junto a la relación entre $\varepsilon_{\mathbf{u}_L}$ y las deformaciones finales requeridas en el sistema permite la introducción de un límite ϵ . Este valor límite permite descartar aquellos vectores que no cumplen con los estándares requeridos, garantizando la selección de vectores óptimos para el proceso de determinación de la CP.

Lo primero que se requiere para calcular $\varepsilon_{\mathbf{u}_L}$ es conocer para el vector \mathbf{u} su correspondiente vector \mathbf{u}_L , lo cual ya fue definido. Una vez conocido este vector se puede definir una forma de medir la diferencia entre ellos.

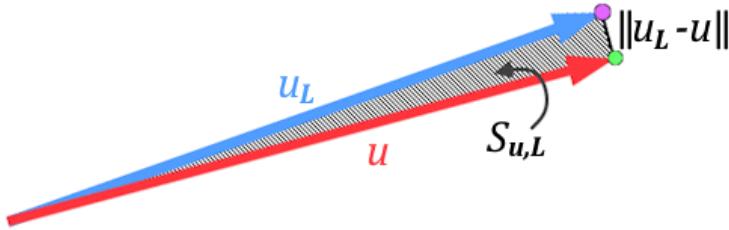


Figura 4.3: $S_{\mathbf{u}_L}$ es la diferencia entre el vector \mathbf{u} y su vector correspondiente en la red L , \mathbf{u}_L

Al conocer \mathbf{u}_L , se puede pensar que este se debe *deformar* hasta coincidir con \mathbf{u} , esta deformación nos permitiría medir su *error*. Así, definimos $S_{\mathbf{u}_L}$ como la suma de las distancias desde cada punto en \mathbf{u}_L y su correspondiente punto en \mathbf{u} , ver la figura 4.3.

Con lo anterior podemos definir $err(\mathbf{u}, \mathbf{u}_L) = \varepsilon_{\mathbf{u}_L}$ como el promedio de $S_{\mathbf{u}_L}$:

$$\varepsilon_{\mathbf{u}_L} = \frac{S_{\mathbf{u}_L}}{\|\mathbf{u}_L\|}. \quad (4.3)$$

Sean $\mathbf{u} = (u_x, u_y)$ y $\mathbf{u}_L = (v_x, v_y)$, entonces, para cada punto $\mathbf{v}_t = (t * v_x, t * v_y) \in \mathbf{u}_L$ corresponde el punto $\mathbf{u}_t = (t * u_x, t * u_y) \in \mathbf{u}$ con $t \in [0, 1]$. Así:

$$S_{\mathbf{u}_L} = \int_0^1 dist(\mathbf{v}_t, \mathbf{u}_t) dt$$

La función “*dist*” es una función que dados 2 vectores regresa la distancia entre ellos, esta es igual a la longitud del vector resultante de los vectores que recibió, osea, dados $\mathbf{a} = (a_1, a_2)$ y $\mathbf{b} = (b_1, b_2)$, entonces:

$$dist(\mathbf{a}, \mathbf{b}) = ||(\mathbf{b} - \mathbf{a})|| = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2}$$

Por lo que tenemos que:

$$\begin{aligned} S_{\mathbf{u}_L} &= \int_0^1 \sqrt{(t * u_x - t * v_x)^2 + (t * u_y - t * v_y)^2} dt \\ &= \int_0^1 \sqrt{t^2 ((u_x - v_x)^2 + (u_y - v_y)^2)} dt \\ &= \int_0^1 t \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2} dt \\ &= \int_0^1 t \|\mathbf{u} - \mathbf{u}_L\| dt \\ &= \|\mathbf{u} - \mathbf{u}_L\| \int_0^1 t dt \end{aligned}$$

Así:

$$S_{\mathbf{u}_L} = \frac{\|\mathbf{u} - \mathbf{u}_L\|}{2} \quad (4.4)$$

Sustituyendo $S_{\mathbf{u}_L}$ de la ecuación (4.4) en la ecuación (4.3), se obtiene:

$$err(\mathbf{u}, \mathbf{u}_L) = \varepsilon_{\mathbf{u}_L} = \frac{\|\mathbf{u} - \mathbf{u}_L\|}{2\|\mathbf{u}_L\|} \quad (4.5)$$

De esta forma se obtiene $\varepsilon_{\mathbf{u}L}$, que será el error asociado al vector \mathbf{u} con respecto a la red L . Este valor de error indicará la elegibilidad para este vector como VP del sistema estudiado.

La definición de $\varepsilon_{\mathbf{u}L}$ en la ecuación (4.5) cumple los requisitos arriba estipulados para el error que se busca:

- Por la forma en que se definió, $\varepsilon_{\mathbf{u}L}$ está relacionada a la deformación requerida en la red L para mantener la periodicidad del sistema.
- $\varepsilon_{\mathbf{u}L}$ tiene una solución única para cualquier vector valido para ser VP. Dado que un vector nulo no puede ser elegible como VP, entonces siempre se cumple que $\|\mathbf{u}\| \neq 0$, lo que evita indeterminaciones en $\varepsilon_{\mathbf{u}L}$.
- El valor de $\varepsilon_{\mathbf{u}L}$ es siempre no negativo, dado que $\|\mathbf{u} - \mathbf{u}_L\| \geq 0$ y $\|\mathbf{u}\| > 0$.
- En el resultado óptimo $\mathbf{u} = \mathbf{u}_L$ y $\varepsilon_{\mathbf{u}L} = 0$. Entre mayor sea la diferencia entre \mathbf{u} y \mathbf{u}_L mayor será el valor de $\varepsilon_{\mathbf{u}L}$.

4.1.3 Algoritmo

Ya definidas las herramientas utilizadas por el el programa para resolver este primer paso, se tiene el siguiente algoritmo:

Creación de lista **R**

(4.6)

0. Inicia para un sistema S .
1. Recibe n y ε .
2. Inicializa la lista R con todos los VTs de la “capa sustrato” ($S.redes[0]$) localizados dentro del área de búsqueda. Inicializa un contador i desde 1: $i = 1$.
3. Depura la lista R para la red L . Sea L la red en la capa i de S , entonces para cada vector $\mathbf{u} \in R$:
 - Identifica \mathbf{u}_L , el VT de L más parecido a \mathbf{u} .
 - Calcula el error entre \mathbf{u} y \mathbf{u}_L , $err(\mathbf{u}, \mathbf{u}_L)$.
 - Si no se cumple que $err(\mathbf{u}, \mathbf{u}_L) < \varepsilon$, entonces elimina \mathbf{u} de R .
4. Si $i \geq (len(S.redes) - 1)$, entonces salta al paso 6.
5. Incrementa el contador i en 1 y regresa al paso 3.
6. Regresa R como la lista de posibles VPs para S con los valores para n y ε dados.
7. Termina.

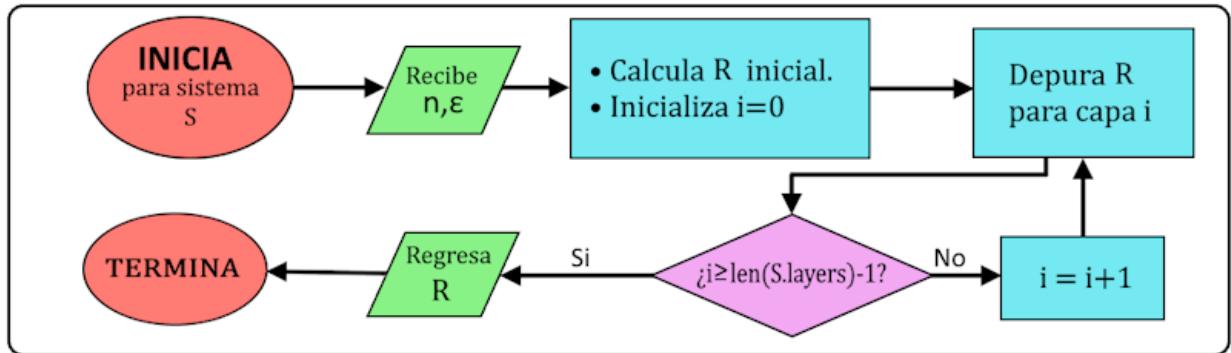


Figura 4.4: Diagrama de flujo para el algoritmo que calcula candidatos a VPs

En el programa, `searchLP(rangeOfSearch, epsilon)` es la instrucción encargada de hacer este cálculo, a `rangeOfSearch` le corresponde el valor de n y a `epsilon` el de ε .

Al finalizar el cálculo de la lista **R**, si **R** esta vacía significa que en toda el área de búsqueda no existe ningún \mathbf{u} en los VTs de la capa sustrato del sistema. En este caso

los vectores correspondientes \mathbf{u}_L en todas las demás capas, presentan un error menor al límite dado. Así que para obtener una lista *no vacía* se requiere aumentar el área de búsqueda incrementando el valor de n , o permitir un error mayor incrementando el valor de ε .

Al analizar la complejidad de este algoritmo, dado que el cálculo de los vectores u_L para los vectores u y el cálculo del error entre estos se obtienen respectivamente de las ecuaciones (4.2) y (4.5), su cálculo se obtiene en tiempo constante ($O(1)$). Así, es importante mencionar que el peso de la complejidad del algoritmo está determinado por el número de posibles vectores u en R y el número de capas del sistema.

El tamaño de la lista R está acotado por el total de puntos de red de la capa cero de S (capa sustrato), que se encuentran dentro del área de búsqueda. Este total es de $2n \times 2n = 4n^2$, por otro lado si tomamos el número de capas del sistema como c , este será una constante por lo regular pequeña (la mayoría de los sistemas evaluados serán de hasta 3 capas), así que podemos obviarla y evaluar este algoritmo como perteneciente al $O(n^2)$ donde n es referente a la variable que indica el área de búsqueda para este.

Por otro lado, al hacer un análisis más detallado del tamaño final de la lista R , podemos ver que esta solo puede llegar a ser igual a su cota en el caso de que no se descarte ninguno de los PRs iniciales, osea que para todos los PRs de la capa sustrato exista un PR en cada capa para el que su error calculado sea menor al ε dado. Esto solamente ocurre si el valor de ε es muy grande o si todas las redes que conformen el sistema tienen la misma orientación y la CP de la capa sustrato es de misma forma y tamaño que alguna supercelda de cada otra capa del sistema.

Lo anterior ocurre dado que en el paso del algoritmo en que se efectúa la comparación con la red de cada capa, el tamaño de la R inicial usualmente disminuye, ya que existen PRs de la capa sustrato para los que el error con su PR correspondiente a esa capa es mayor que ε , eliminando dichos puntos de la lista R para el análisis de la siguiente capa. El tamaño de esta disminución es mayor entre más distintas sean las redes en el sistema y menor sea el valor ε dado, de esta forma en sistemas con rotaciones y distintas redes de Bravais en cada capa o en análisis con ε muy pequeñas, el tamaño final de R será mucho menor a su cota n^2 .

Con el análisis anterior también podemos concluir que el tamaño de la lista R final tiende a ser inversamente proporcional al número de capas del sistema.

4.2 Cálculo de matrices de transformación

El método en el programa encargado de llevar a cabo esta tarea es `calculateTM()` de la clase “System”, el cual guarda la lista de matrices calculadas como la lista `loMat` de la clase System.

Las MTs indican una transformación lineal que, al aplicarse a los VPs de una red, generan los vectores que conforman una supercelda para dicha red. Debido a esto, el determinante de una MT nos indica cuántas veces es más grande la supercelda relacionada con ella en comparación con la celda primitiva de la red. Por lo tanto, al buscar las superceldas más pequeñas para utilizarlas como celdas primitivas (CP) del sistema, podemos calcular el determinante de las MTs para ordenar estos resultados y seleccionar las superceldas más pequeñas.

Dado que ya se tiene R , una lista de VT válidos en todas las capas del sistema, se puede generar MTs al tomar pares de vectores en R , sin embargo no todas los pares de vectores dan MTs que generen superceldas que puedan ser usadas como CP para el sistema. Por ejemplo si los vectores son colineales. Para descartar estos casos, también es útil calcular el determinante de la MT. Si el determinante es cero, entonces la MT es desechada y se continúa con otra.

Al momento de calcular el determinante también se puede forzar a que los vectores mantengan el mismo “sentido”, así, si el determinante es negativo los vectores se intercambian, de manera que el nuevo determinante sea positivo, dejando resultados coherentes y de forma similar.

Por otro lado, aprovechando que los resultados obtenidos son los más “pequeños”, se puede acotar el número de respuestas para limitar el costo computacional del cálculo. El límite es dado por la variable de la clase sistema llamada `MaxNumM`, la cual por defecto vale 10 pero puede ser cambiada por el usuario. Esta variable va a limitar el tamaño de la lista `loMat`, acotando el número de MTs que se mostrarán al usuario por respuestas.

Una vez creada la lista `loMat`, utilizando las Ecs. 4.2 podemos obtener las MTs para cada capa correspondiente a las MTs guardadas en ella.

El algoritmo usado para calcular esta lista de matrices es el siguiente:

Creación de lista de MTs *loMat*

(4.7)

0. Inicia para un sistema *S*.
1. Recibe lista de vectores *R* y la variable *numOfM*
2. Inicializa *loMat* con una la lista vacía. Inicializa contador *i* = 0.
3. Inicializa contador *j* = *i* + 1.
4. Si *j* ≥ |*R*|, entonces salta al paso 10.
5. Crea la matriz $M_{i,j}$ a partir de los vectores *R*[*i*] y *R*[*j*].

6.

$$\text{biggestM} = \begin{cases} \infty & \text{si } loMat \text{ es vacía} \\ \max\{\det(M) : M \in loMat\} & \text{si } loMat \text{ no es vacía} \end{cases}$$

Si $\det(M_{i,j}) \geq \text{biggestM}$ salta al paso 9.

7. Si $M_{i,j}$ es una MT válida y es nueva para *loMat*, la agrega de forma ordenada a la lista.
8. Si $\text{len}(loMat) > \text{numOfM}$, *loMat* se corta hasta tener *numOfM* elementos.
9. Incrementa el contador *j* en 1, regresa al paso 4.
10. Incrementa el contador *i* en 1.
11. Si *i* < |*R*| regresa al paso 3.
12. Regresa *loMat* como la lista de las “*numOfM*” MT que generan las celdas periódicas más pequeñas para *S*.
13. Termina.

En el paso 7 del algoritmo se señala que se agrega la matriz $M_{i,j}$ calculada a la lista *loMat* si esta es una MT válida y si, además, es nueva para la lista. Lo primero es referente a lo señalado antes, si el determinante de la matriz es cero implicaría que los vectores utilizados son colineales, entonces la matriz se descarta ya que no hace referencia a una celda válida para la red. Lo segundo es un poco más complicado, pero igual de importante, ya que aquí se descartan matrices que sean una rotación exacta de alguna matriz que ya esté contemplada en *loMat*.

Otras matrices que se descartan en ese mismo paso son aquellas que generen celdas muy poco convencionales, como aquellas que tienen un ángulo muy cerrado entre los vectores que la conforman (por defecto se descartan matrices que generen celdas con

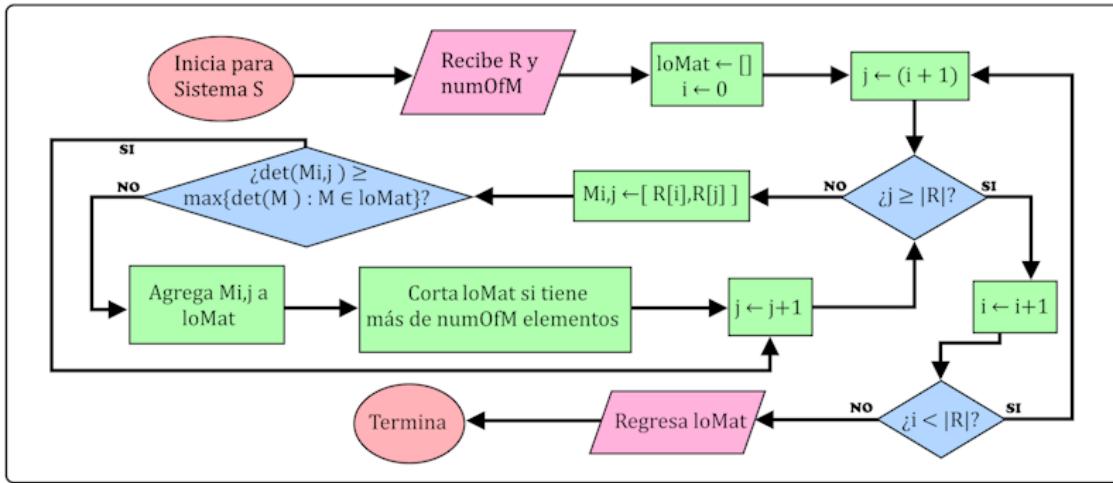


Figura 4.5: Diagrama de flujo para el algoritmo que calcula la lista loMat

menos de 20° de ángulo interno), o que la proporción entre el tamaño de los vectores sea mayor a 10.

En el análisis de complejidad de este algoritmo se puede ver que el mayor peso de esta se lleva en el proceso de agregar las matrices de forma ordenada por el valor de su determinante, después de ser evaluadas, en la lista *loMat*. Entonces la complejidad de este algoritmo se acota solamente por el número de matrices evaluadas, que para una lista *R* de tamaño *m* se tienen un número de matrices distintas igual a $\frac{m(m-1)}{2}$. Esto es debido a que el proceso para mantener el orden de la lista resultante al agregar una nueva matriz a esta se acota por la constante *numOfM*.

Recordando que el tamaño de la lista *R* está acotado por el cuadrado de la constante *n* que determina el área de búsqueda para el algoritmo general, se puede acotar entonces este algoritmo en $O(n^4)$, aunque, como ya se señaló, en la ejecución normalmente el tamaño de la lista *R* es mucho menor a n^2 , por lo que esta cota queda muy por encima de lo que se ve al ejecutar el código.

Otra medida de *optimización* se da en el caso de que todas las redes que los conforman al sistema evaluado sean redes hexagonales, en estos casos, dadas las simetrías de los sistemas, se espera que la supercelda usada como CP, también tenga una red de Bravais hexagonal, por lo que se puede evitar calcular todas las combinaciones de pares de elementos de la lista *R* al momento de crear las matrices a evaluar. Así, si el sistema está formado solo por redes hexagonales se sigue el siguiente método para calcular una MT que dé como resultado una celda hexagonal para cada VT en *R*:

Sea $V = x\mathbf{a} + y\mathbf{b}$ un VT guardado en *R*, entonces a este vector se le asocia la MT, *T*,

tal que:

$$T = \begin{pmatrix} x & -y \\ y & x-y \end{pmatrix}$$

De esta forma, cuando se evalúen sistemas formados por solo redes hexagonales (sin importar que sus constantes de red sean distintas y/o estén rotadas), el número de matrices que se evaluarán es igual al tamaño de R .

4.3 Cálculo de la matriz de deformación

Salvo casos especiales, algunas capas del sistema requerirán sufrir una deformación para mantener la periodicidad completa del sistema. Esta deformación no puede ser muy grande y está limitada directamente por el valor que se le dio a la variable ε en el momento de calcular los candidatos a VPs para el sistema.

Esta deformación se ve reflejada en matrices por las que requieren ser multiplicados los VPs de cada capa respectivamente, para que al aplicar su correspondiente MT, el resultado coincidan perfectamente con los vectores seleccionados a usar como VPs para el sistema. A esta matriz se le llamará matriz de deformación (MD).

Sea:

- V_s la matriz de VPs de la red en la capa sustrato del sistema (capa 0).
- V_i la matriz de VPs de la red en la capa i .
- T la MT que generará la CP para el sistema.
- T_i la MT propia para la capa i del sistema.

Entonces la MD para la capa i será D_i , la cual es una matriz que cumple la siguiente igualdad:

$$V_s T = (V_i D_i) T_i$$

Sabemos que V_i y T_i son matrices con determinante distinto de cero, por lo tanto tienen inverso, así obtenemos la siguiente ecuación:

$$D_i = V_i^{-1} (V_s T) T_i^{-1} \quad (4.8)$$

Los nuevos VPs para la red del sistema en la capa i se obtendrán de las columnas de la matriz $V'_i = V_i D_i$, lo cual generará un ajuste en la CP de la red. El cambio experimentado por un VP \mathbf{u} tras la deformación se puede describir de manera más clara

utilizando las variables ΔM_u y ΔA_u , donde ΔM_u representa el porcentaje de variación en la magnitud del vector respecto a su valor original, y ΔA_u indica el cambio en el ángulo de su dirección.

Así, sean $\mathbf{u} = m (\cos(t)\hat{i} + \sin(t)\hat{j})$ y $\mathbf{u}' = m' (\cos(t')\hat{i} + \sin(t')\hat{j})$ un vector original y el resultado de este tras sufrir una deformación, entonces, definimos ΔM_u y ΔA_u de la siguiente manera:

$$\Delta M_u = \frac{(m' - m)}{m} \times 100 \quad \Delta A_u = (t' - t) \quad (4.9)$$

Para contabilizar el cambio producido en la red se utiliza el siguiente método:

Sean a, b los VPs originales de la red y a', b' los VPs deformados, entonces tomando los vectores $P_k = i_k a + j_k b$, $Q_k = i_k a' + j_k b'$ con $i_k, j_k \in [0, 1]$, $k \in \{1, 2, \dots, n\}$, tendremos n parejas de vectores que van del origen a puntos dentro del área delimitada por las CPs original y deformada de la red respectivamente, entonces se define el grado de distorsión de la red DD como:

$$DD = \sum_{k=1}^n \left(\frac{\text{err}(P_k, Q_k)}{n} \right) \quad (4.10)$$

En donde se hace uso de la función err , definida en la ecuación (4.5), para obtener una aproximación al error promedio (la distorsión promedio) generado tras la deformación.

Cada MT que haga referencia a una posible solución al sistema va a implicar deformaciones para cada capa del sistema (la cual puede ser nula), estas deformaciones en las redes de cada capa las podemos calificar con su respectiva DD , por construcción, la deformación requerida por la capa sustrato es nula, lo que implica siempre tener un $DD = 0$ para esta capa.

A la suma de los DD en todas las capas del sistema inducidos por una MT la denominamos DD total (DD_t). De esta manera podemos calificar cada MT por el grado de deformación que esta implica, entre menor sea el valor del DD_t mejor opción será la MT como resultado, siendo un resultado $DD_t = 0$ el óptimo indicando una deformación nula en todo el sistema.

Tras diversas pruebas en ejecución, se observó que pueden catalogarse como *resultados muy buenos* aquellos cuyo DD_t sea menor a $0,02 * (c - 1)$ con c el número de capas del sistema, ya que esto significa que, en promedio, el valor del “error” correspondiente los VPs de todas las capas sobre la capa sustrato del sistema será menor al 4 %.

Tener esta “calificación” para cada resultado permite obtener un resultado que optimice tanto el tamaño de la CP (esto gracias a que las matrices en *loMat* ya son las MTs

más pequeñas posibles con los parámetros dados), así como la deformación de las capas del sistema. Esta es la matriz que el programa recomendará como respuesta al usuario si este decide no escoger una MT manualmente.

Para ejemplificar todo lo anterior, se tomará un sistema S , compuesto de tres capas: *grafeno*, *grafeno rotado 13,5º* y *fosforeno negro*. Para este ejemplo se decide usar la MT:

$$T = \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}$$

Esta MT induce a que los VPs para el sistema sean los vectores $\mathbf{u} = 3a_0 - 2b_0$ y $\mathbf{v} = 9a_0 + 2b_0$, donde $a_0 = (2,44, 0,0)$ y $b_0 = (-1,22, 2,1131)$ son los VPs de la primer capa de grafeno (la capa sustrato del sistema). De esta forma se tendría que:

$$\mathbf{u} = (9,76, -4,2262), \mathbf{v} = (19,52, 4,2262)$$

Al calcular los VTs para la red de grafeno rotado más parecidas a estos vectores se obtienen $\mathbf{u}_1 = 2a_1 - 3b_1$ y $\mathbf{v}_1 = 8a_1$. Aquí los VPs del grafeno rotado son $a_1 = (2,3726, 0,5696)$ y $b_1 = (-1,6796, 1,7699)$, así:

$$\mathbf{u}_1 = (9,7839, -4,1705), \mathbf{v}_1 = (18,9807, 4,5569)$$

De forma similar para la red de fosforeno, $\mathbf{u}_2 = 3a_2 - b_2$ y $\mathbf{v}_2 = 6a_2 + b_2$. En este caso los VPs de la red son $a_2 = (3,2601, 0,0)$ y $b_2 = (0,0, 4,347)$ haciendo que:

$$\mathbf{u}_2 = (9,7804, -4,3470), \mathbf{v}_2 = (19,5608, 4,3470)$$

Con este ejemplo se puede ver que $\mathbf{u} \neq \mathbf{u}_1 \neq \mathbf{u}_2$ y $\mathbf{v} \neq \mathbf{v}_1 \neq \mathbf{v}_2$, por lo que si no se aplica una deformación a las 2 capas superiores, estas perderán su periodicidad.

Para este caso:

$$V_s = \begin{pmatrix} 2,4400 & -1,2200 \\ 0,0000 & 2,1131 \end{pmatrix}, T = \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}$$

$$V_1 = \begin{pmatrix} 2,3726 & -1,6796 \\ 0,5696 & 1,7699 \end{pmatrix}, T_1 = \begin{pmatrix} 2 & 8 \\ -3 & 0 \end{pmatrix}$$

$$V_2 = \begin{pmatrix} 3,2601 & 0,0000 \\ 0,0000 & 4,347 \end{pmatrix}, T_2 = \begin{pmatrix} 3 & 6 \\ -1 & 1 \end{pmatrix}$$

Utilizando la ecuación 4.8 obtenemos las MDs correspondientes para cada capa.

$$D_0 = \begin{pmatrix} 2,4400 & -1,2200 \\ 0,0000 & 2,1131 \end{pmatrix}^{-1} \left(\begin{pmatrix} 2,4400 & -1,2200 \\ 0,0000 & 2,1131 \end{pmatrix} \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix} \right) \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix}^{-1}$$

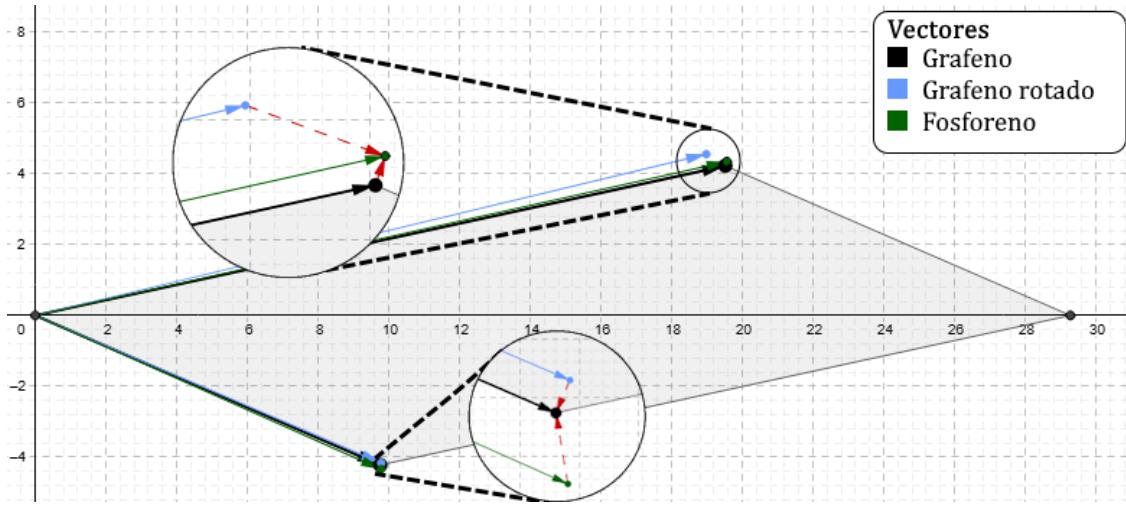


Figura 4.6: Celda primitiva tentativa para un sistema formado por grafeno, grafeno rotado $13,5^\circ$, y fosforeno negro. La celda esta definida por los VTs de la primer capa de grafeno (vectores negros), se muestran también los VT más parecidos a estos en las capas de grafeno rotado (azul) y fosforeno (verde). Se puede apreciar que estos últimos no coinciden perfectamente con la celda primitiva por lo que requieren una deformación para empatar las celdas.

$$D_1 = \begin{pmatrix} 2,3726 & -1,6796 \\ 0,5696 & 1,7699 \end{pmatrix}^{-1} \left(\begin{pmatrix} 2,4400 & -1,2200 \\ 0,0000 & 2,1131 \end{pmatrix} \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix} \right) \begin{pmatrix} 2 & 8 \\ -3 & 0 \end{pmatrix}^{-1}$$

$$D_2 = \begin{pmatrix} 3,2601 & 0,0000 \\ 0,0000 & 4,347 \end{pmatrix}^{-1} \left(\begin{pmatrix} 2,4400 & -1,2200 \\ 0,0000 & 2,1131 \end{pmatrix} \begin{pmatrix} 3 & 9 \\ -2 & 2 \end{pmatrix} \right) \begin{pmatrix} 3 & 6 \\ -1 & 1 \end{pmatrix}^{-1}$$

Obteniendo:

$$D_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad D_1 = \begin{pmatrix} 1,00968 & 0,01524 \\ -0,02647 & 0,99001 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0,99792 & 0,00000 \\ 0,00000 & 0,97220 \end{pmatrix}$$

Determinando que a excepción de la capa cero del sistema, las demás capas sí deben tener una deformación. Al aplicar esta deformación a los VPs de cada capa obtenemos unos nuevos VPs, los cuales sí mantienen la periodicidad en todo el sistema con la PC elegida.

$$V'_1 = \begin{pmatrix} 2,4400 & -1,6267 \\ 0,5283 & 1,7609 \end{pmatrix} \quad V'_2 = \begin{pmatrix} 3,253318992 & 0 \\ 0 & 4,2261534 \end{pmatrix}$$

Estas deformaciones sufridas por las capas 1 y 2 del sistema, al aplicar la ecuación 4.10, usando 100 pares de vectores en ella, dan como resultado $DD_1 = 0,021002$ y $DD_2 = 0,009611$, con un $DDt = 0,030613$, este valor nos indica que el resultado es lo suficientemente bueno ya que:

$$DDt < 0,02 * (3 - 1)$$

Lo que cumple la condición antes mencionada.

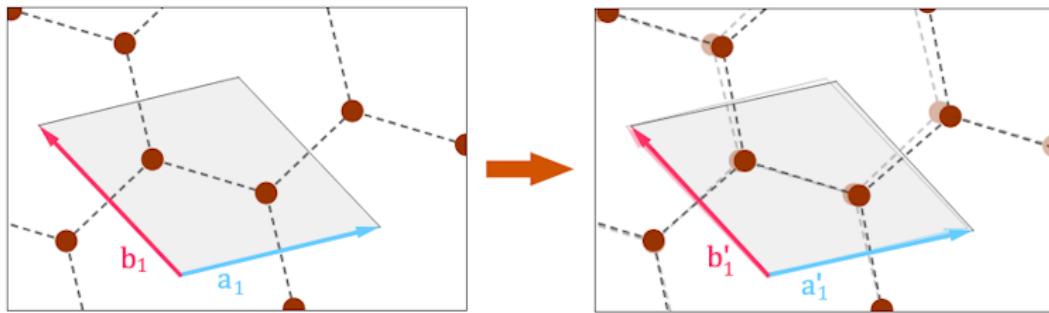


Figura 4.7: Deformación efectuada en la capa de grafeno rotada para el sistema definido. Esta deformación induce un cambio en sus VPs a y b de manera que: $\Delta M_a = 2,3\%$, $\Delta M_b = 1,7\%$, $\Delta A_a = -1,28^\circ$ y $\Delta A_b = -0,76^\circ$. Esta deformación tiene un $DD = 0,021002$.

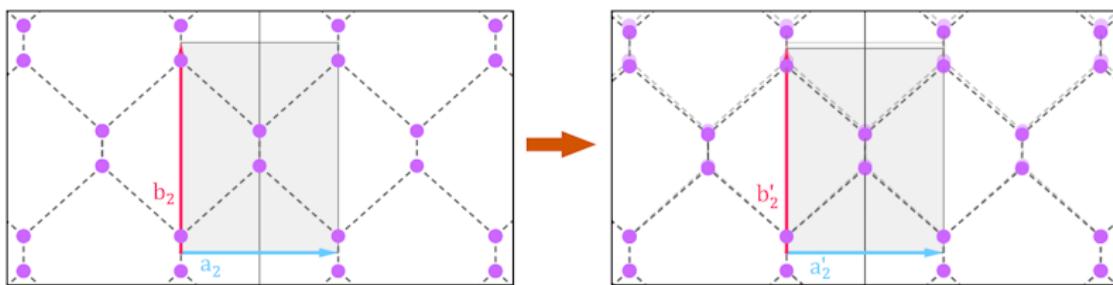


Figura 4.8: Deformación efectuada en la capa de fosforeno negro para el sistema definido. En este caso, la deformación induce un cambio en los VPs a y b de manera que: $\Delta M_a = 0,2\%$, $\Delta M_b = 2,8\%$, no cambia su dirección por lo que $\Delta A_a = \Delta A_b = 0^\circ$. Esta deformación tiene un $DD = 0,009611$.

La información sobre la deformación requerida por las redes del sistema para cada MT en *loMat*, que es proporcionada por el método *ShowTMs* mencionado en la subsección anterior, se trata de la MD correspondiente a cada red, las cuales están en la columna “**Deformation**” y las ΔM y ΔA que dichas deformaciones generan en sus VPs, estas en la columna “**Distortion δ//θ:**”, donde δ corresponde al ΔM y ϑ al ΔA . Por último, el valor **DD** expresado para la tabla de cada MT corresponde al **DDt** del sistema para dicha MT. Estos datos pueden verse en la figura 3.5, ubicada en la sección “*Algoritmo principal*” de este mismo capítulo, la cual muestra ejemplos de las tablas desplegadas por dicho método.

4.4 Cálculo y muestra de la primera zona de Brillouin

Como ya se ha señalado, uno de los resultados extras dados por el programa, es una imagen de la primer zona de Brillouin (PZB) de las celdas primitivas para cada capa del sistema, así como una imagen de la PZB del sistema completo. Estos datos pueden ser de ayuda para el usuario ya que proporciona información crucial sobre las trayectorias que conectan puntos de alta simetría para la determinación de relaciones de dispersión electrónicas y fonónicas.

La PZB de cada red cristalina 2D corresponderá a un paralelogramo dependiendo de la simetría de la red de Bravais a la que pertenece. Para calcular la PZB se calcula la celda de Wigner-Seitz en el espacio recíproco usando los vectores recíprocos de la red.

Así la PZB corresponderá al polígono convexo delimitado por las mediatrices entre el origen y los puntos de la red recíproca. Para definir este polígono debemos conocer los puntos donde caen sus vértices, que serán donde se crucen las mediatrices que tocan al polígono.

Para calcular estos puntos, el programa hace uso de una modificación del *algoritmo de optimización Simplex*[23]² con el método `calcVerticesFBZ` en el archivo `basics.py`. El método Simplex puede adaptarse para recorrer el perímetro de una figura convexa en 2D debido a que su algoritmo se basa en el movimiento sistemático entre vértices de un poliedro convexo a través de sus aristas, optimizando una función objetivo en cada paso. En el caso de una figura convexa bidimensional, como en el caso de la PZB que buscamos, este principio es directamente aplicable, ya que las aristas y vértices del polígono convexo representan el límite de la región factible. Según Bazaraa, Jarvis y Sherali [24], el método Simplex utiliza un enfoque estructurado para transitar por las aristas de un poliedro, lo que permite adaptar su estrategia para recorrer de manera ordenada los vértices de una figura convexa, garantizando un tránsito continuo por su perímetro.

En esta modificación del algoritmo Simplex que implementamos, efectuamos las operaciones sobre una matriz dada por las ecuaciones de las mediatrices calculadas. En el programa hacemos uso de las mediatrices entre el origen y los 8 puntos de la red recíproca más cercanos para encontrar la primer frontera; y a partir de ahí recorrer cada arista del polígono encontrando los vértices, terminando así cuando el polígono se cierra.

Para ejemplificar el funcionamiento del algoritmo, se irá resolviendo un ejemplo, en donde se mostrará lo que geométricamente está ocurriendo en cada paso.

²También conocido como método de Nelder-Mead o método *Amoeba* entre otros.

Sea L una red cristalina 2D oblicua con vectores primitivos $a = (2,5,0,0)$ y $b = (-1,75,3,0)$ a la que calcularemos su PZB. El primer paso es identificar los 8 puntos de la red, en el espacio recíproco, más cercanos al origen, los cuales en este caso son:

$$\begin{array}{ll} V_1 = (0,0000, -2,0944) & V_2 = (0,0000, 2,0944) \\ V_3 = (-2,5133, 0,6283) & V_4 = (2,5133, -0,6283) \\ V_5 = (-2,5133, -1,4661) & V_6 = (2,5133, 1,4661) \\ V_7 = (-2,5133, 2,7227) & V_8 = (2,5133, -2,7227) \end{array}$$

Con estos puntos v_i calculamos las rectas mediatrices m_i entre ellos y el origen (figura 4.9).

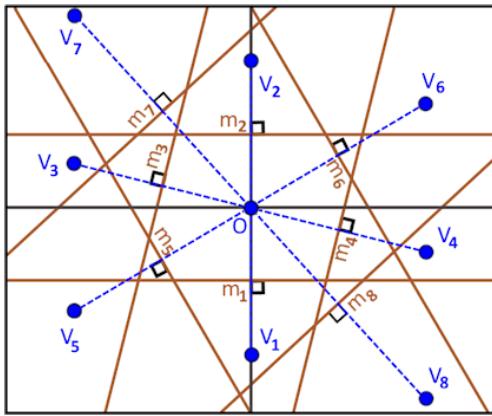


Figura 4.9: PRs más cercanos al origen y sus mediatrices con este.

A partir de estas rectas creamos una tabla de 11 columnas por 8 filas. Con las primeras 8 columnas crearemos una matriz identidad de 8×8 . Mientras que las columnas 9, 10 y 11 serán los valores a , b y c de las ecuaciones de cada una de las rectas m_i , donde cada fila i corresponderá a una recta m_i ; tal y como se ve en la tabla 4.1.

$$\begin{array}{ll} m_1 : 0,00x - 4,19y = 4,39 & m_2 : 0,00x + 4,19y = 4,39 \\ m_3 : -5,03x + 1,26y = 6,71 & m_4 : 5,03x - 1,26y = 6,71 \\ m_5 : -5,03x - 2,93y = 8,47 & m_6 : 5,03x + 2,93y = 8,47 \\ m_7 : -5,03x + 5,45y = 13,73 & m_8 : 5,03x - 5,45y = 13,73 \end{array}$$

En esta tabla etiquetaremos las primeras 8 columnas para hacer referencia a las rectas m_i mientras que las columnas 9 y 10 corresponderán a las rectas $x = 0$ y $y = 0$ (aquí, al estar en el espacio recíproco deberíamos trabajar con la base recíproca x' , y' , sin embargo por comodidad se llamarán como x y y). A partir de esta tabla se iniciará el algoritmo.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-4.19	4.39
m_2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.19	4.39
m_3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	-5.03	1.26	6.71
m_4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	5.03	-1.26	6.71
m_5	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	-5.03	-2.93	8.47
m_6	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	5.03	2.93	8.47
m_7	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	-5.03	5.45	13.73
m_8	0.0	0.0	0.0	0.0	0.0	0.0	1.0	5.03	-5.45		13.73

Cuadro 4.1: Estado inicial de la tabla

El primer paso es ubicarnos en el perímetro del polígono para movernos por él y encontrar un primer vértice. Iniciamos encontrando la intersección del perímetro con el eje x en el lado positivo, para esto, en la tabla buscamos “*sacar*” una de las filas m_i y “*meter*” en su lugar una fila correspondiente a la columna x .

Lo que hacemos es fijarnos en las filas con valores mayores a cero en la columna x ; en estas filas dividimos el valor en la columna c entre el valor en la columna x y nos tomamos como *pivote* a la celda en la columna x de la fila que arrojó el menor resultado.

En este caso, las filas en que nos fijamos son m_4 , m_6 y m_8 que son las únicas con valores mayores a cero en la columna x , al hacer las operaciones obtenemos $\frac{6,71}{5,03}$ para m_4 , $\frac{8,47}{5,03}$ para m_6 y $\frac{13,73}{5,03}$ para m_8 , por lo que la fila con el menor valor es m_4 , así tomamos como pivote la casilla $[m_4, x]$.

Teniendo localizado el pivote, debemos “*meter*” la fila x , para esto debemos hacer operaciones de tal manera que al final la columna x tenga ceros en todas sus casillas excepto en la casilla pivote, donde debemos tener un valor de 1, llegando así a una nueva tabla (Tabla 4.2).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-4.19	4.39
m_2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.19	4.39
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
x	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	1.0	-0.25	1.34
m_5	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	-4.19	15.18
m_6	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	4.19	1.75
m_7	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	4.19	20.44
m_8	0.0	0.0	0.0	-1.0	0.0	0.0	0.0	1.0	0.0	-4.19	7.02

Cuadro 4.2: Tabla con una fila correspondiente a x .

Las operaciones que se siguen para realizar el proceso de *sustituir* una fila **a** por una nueva fila **b** en una matriz de valores reales, utilizando como *pivote* el valor en la posición $[a, b] = p$, son los siguientes:

1. Normalización de la fila pivote:

- Se dividen todos los elementos de la fila **a** por el valor del pivote p , realizando la operación $\mathbf{a} = \mathbf{a}/p$.
- Se renombra la fila **a** como **b**. Con esto, la fila **b** ocupa el lugar de la fila original **a**, y el valor en la posición de pivote (ahora $[b, b]$) se convierte en 1.

2. Reducción de las demás filas:

- Para cada fila **c** distinta de la fila **b** ($\mathbf{c} \neq \mathbf{b}$), se realiza la operación $\mathbf{c} = \mathbf{c} - q_c \mathbf{b}$, donde q_c es el valor de la casilla $[c, b]$ en la fila **c**. Esta operación hace que el valor en la posición $[c, b]$ de cada fila **c** (distinta de **b**) se convierta en 0.

Siguiendo estos pasos, se obtiene una nueva fila **b** en la posición de la fila **a**, logrando que toda la columna correspondiente a **b** tenga ceros, excepto en la posición $[b, b]$, que contiene el valor 1 resultante de la normalización de la fila **b**.

Lo que se hizo al *meter* la fila x , es equivalente a lanzar un rayo desde el origen en dirección del eje X, obtener las intersecciones de este con las mediatrices dibujadas y quedarnos con la más próxima al origen³, en el caso del ejemplo fue la intersección del eje X con la recta m_4 (figura 4.10), posicionándonos en un punto del perímetro de la PZB de la red.

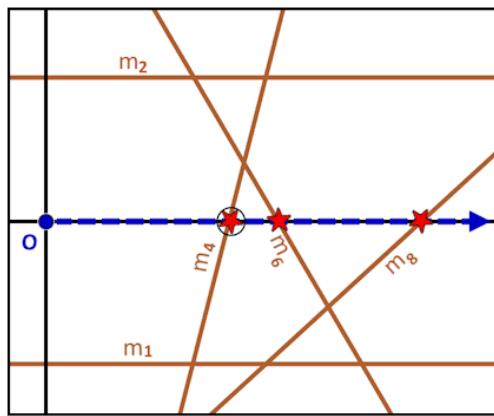


Figura 4.10: Representación visual del primer paso del algoritmo, en este se encuentran las intersecciones del rayo a partir del origen en dirección del eje X y las mediatrices calculadas.

Ahora, para encontrar el primer vértice del polígono debemos meter a y en las filas. Para esto hacemos un paso similar al anterior, buscando un pivote, pero ahora en la columna y en lugar de la columna x , siendo en esta ocasión la casilla $[m_6, y]$ la que tomamos

³Esta interpretación es tomada directamente de la interpretación geométrica para el inicio del método Simplex, donde nos posicionamos de inicio en un punto que es una solución básica factible, en este caso, el punto máximo con $y=0$ dentro del polígono convexo delimitado por las mediatrices trazadas.

de pivote. Ejecutamos las operaciones para meter la fila y , obteniendo una nueva tabla (Tabla 4.3).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
m_2	0.0	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	2.63
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
x	0.0	0.0	0.0	0.14	0.0	0.06	0.0	0.0	1.0	0.0	1.44
m_5	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
y	0.0	0.0	0.0	-0.24	0.0	0.24	0.0	0.0	0.0	1.0	0.42
m_7	0.0	0.0	0.0	2.0	0.0	-1.0	1.0	0.0	0.0	0.0	18.69
m_8	0.0	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	8.77

Cuadro 4.3: Tabla después de meter a x y y en las filas, en este podemos observar que el primer vértice del polígono corresponderá al punto $(1,44, 0,42)$

Con esta tabla obtenemos el valor del punto P_1 , el primer vértice del polígono que representa la PZB de la red, este punto coincidirá con el cruce de las dos rectas que “sacamos” de las filas, en este caso son m_4 y m_6 . Para obtener la posición de este punto debemos ver los valores correspondientes a las filas x y y en la columna c de la última tabla obtenida. En este ejemplo el primer vértice del polígono es $P_1 = (1,44, 0,42)$ (figura 4.11).

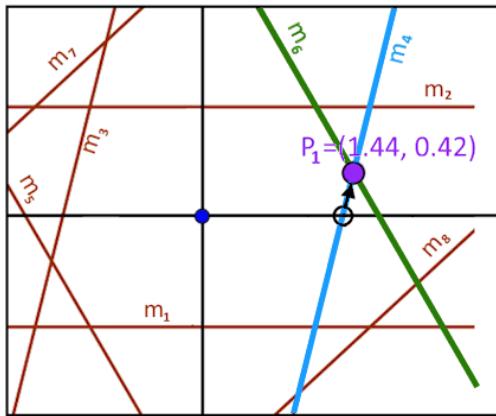


Figura 4.11: Primer vertice encontrado en la intersección de m_4 y m_6

A partir de aquí comienza la segunda fase del algoritmo, en esta se recorren las aristas del polígono encontrando en orden antihorario los demás vértices.

Se inicia buscando “meter” m_k , la penúltima recta que ha salido de las filas. Para esto buscamos la celda pivotе (m_l, m_k) , donde m_l será la fila que “saldrá” y será remplazada por la fila que intentamos “meter”, teniendo cuidado de nunca “sacar” las filas x o y .

La fila m_l será aquella fila entre las filas con valores mayores a cero en la columna m_k con el **menor resultado no negativo** al dividir el valor en la columna c entre su valor en la columna m_k . Si el menor valor no negativo corresponde a las filas x o y optamos por la siguiente mejor opción.

Una vez obtenida la celda pivote, efectuamos las operaciones para meter la fila m_k y sacar la fila m_l , obteniendo una nueva tabla, la cual mostrará la posición del punto $P_i = ([x, c], [y, c])$ el corresponderá a una nueva arista del polinomio.

Repetiremos esto hasta que $P_i = P_1$, donde sabremos que hemos recorrido todo el perímetro y encontrado todos los vértices del polígono.

En el ejemplo, la primer fila que buscamos meter es la correspondiente a la recta m_4 , aquí las filas candidatas a salir son m_2, m_3, x y m_7 , teniendo como resultados de dividir c/m_4 : $\frac{2,63}{1} = 2,63$, $\frac{13,42}{1} = 13,42$, $\frac{1,44}{0,14} = 10,29$ y $\frac{18,69}{2} = 9,35$ respectivamente. Indicando entonces que el pivote será la celda $[m_2, m_4]$. Efectuando las operaciones correspondientes obtenemos una nueva tabla (Tabla 4.4).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.77
m_4	0.0	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	2.63
m_3	0.0	-1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	10.79
x	0.0	-0.14	0.0	0.0	0.0	0.2	0.0	0.0	1.0	0.0	1.07
m_5	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	16.93
y	0.0	0.24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.05
m_7	0.0	-2.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	13.42
m_8	0.0	2.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	14.04

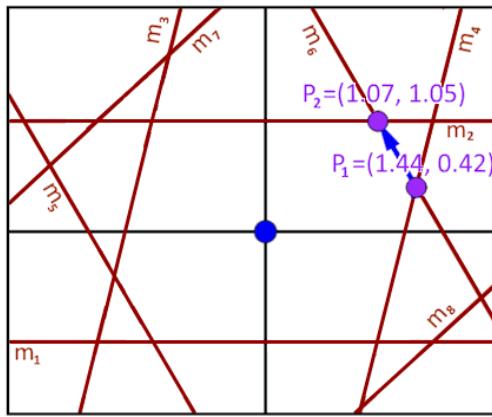
Cuadro 4.4: Tabla después de regresar a m_4 y sacar m_2 de las filas.

De esta tabla obtenemos el punto $P_2 = ([x, c], [y, c]) = (1,07, 1,05)$. En este punto se intersectan las rectas m_6 y m_2 (figura 4.12).

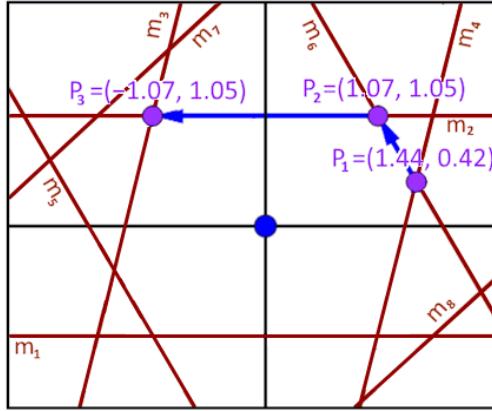
Dado que el punto P_2 es distinto a P_1 repetimos el algoritmo para buscar el siguiente punto.

Ahora intentamos meter la fila m_6 . En este caso la mejor opción de pivote es $[x, m_6]$ pero dado que esto sacaría a x de las filas, se toma la siguiente mejor opción, $[m_3, m_6]$, y se efectúan las operaciones correspondientes para obtener una nueva tabla (Tabla 4.5).

De esta tabla se obtiene el punto $P_3 = (-1,07, 1,05)$, dado que este es distinto a P_1 continuamos (figura 4.13).

Figura 4.12: Segundo vértice encontrado en la intersección de m_6 y m_2

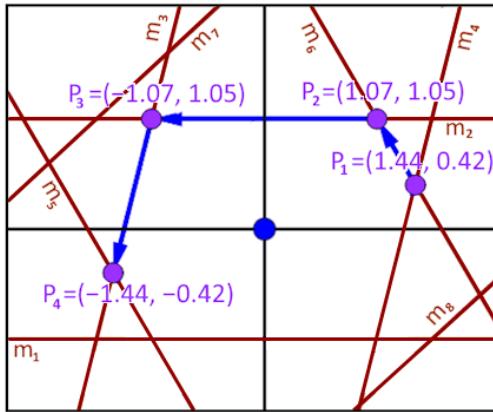
	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.77
m_4	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_6	0.0	-1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	10.79
x	0.0	0.06	-0.2	0.0	0.0	0.0	0.0	1.0	0.0	-1.07	
m_5	0.0	1.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	6.14
y	0.0	0.24	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.05	
m_7	0.0	-1.0	-1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	2.63
m_8	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	24.83

Cuadro 4.5: Tabla después de meter a m_6 y sacar m_3 de las filas.Figura 4.13: Tercer vértice encontrado en la intersección de m_2 y m_3

Buscamos ingresar la fila m_2 , aquí se tiene como mejor opción de pivote $[y, m_2]$, descartamos esta opción y usamos el siguiente mejor, $[m_5, m_2]$ y generamos la nueva tabla (Tabla 4.6).

De esta se obtiene la intersección entre las rectas m_3 y m_5 , $P_4 = (-1.44, -0.42)$ (figura 4.14).

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	0.0	2.63
m_4	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_6	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
x	0.0	0.0	-0.14	0.0	-0.06	0.0	0.0	0.0	1.0	0.0	-1.44
m_2	0.0	1.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	6.14
y	0.0	0.0	0.24	0.0	-0.24	0.0	0.0	0.0	0.0	1.0	-0.42
m_7	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	8.77
m_8	0.0	0.0	2.0	0.0	-1.0	0.0	0.0	1.0	0.0	0.0	18.69

Cuadro 4.6: Tabla después de meter a m_2 y sacar m_5 de las filas.Figura 4.14: Cuarto vértice encontrado en la intersección de m_3 y m_5

Continuamos el algoritmo un par de veces más, metiendo m_3 y sacando m_1 , después metiendo m_5 y sacando m_4 , obteniendo los puntos $P_5 = (-1,07, -1,05)$ y $P_6 = (1,07, -1,05)$.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_5	-1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	10.79
m_6	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
x	-0.06	0.0	0.0	0.2	0.0	0.0	0.0	0.0	1.0	0.0	1.07
m_2	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.77
y	-0.24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	-1.05
m_7	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	24.83
m_8	-1.0	0.0	0.0	-1.0	0.0	0.0	0.0	1.0	0.0	0.0	2.63

Cuadro 4.7: Tabla después de efectuar varias repeticiones del algoritmo y haber obtenido los puntos $P_1, P_2, P_3, P_4, P_5, P_6$.

Tenemos ahora la Tabla 4.7, donde el pivote es (m_1, m_6) , al sacar m_6 para meter m_1 y hacer las operaciones adecuadas obtenemos una nueva Tabla 4.8, la cual nos señala como nueva arista el punto $P_7 = (1,44, 0,42)$. Pero este es igual a P_1 , por lo que ya no admitimos a P_7 como nuevo vértice y terminamos el algoritmo, teniendo que la PZB de la red corresponderá al polígono $[P_1, P_2, P_3, P_4, P_5, P_6]$ (figura 4.15).

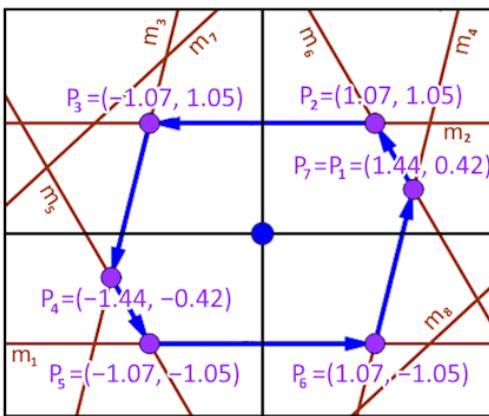


Figura 4.15: Todos los vértices del polígono que forma la PZB y su recorrido en la búsqueda, comenzando en P_1 y terminando con P_7 al ser el mismo punto.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	x	y	c
m_1	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	13.42
m_4	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	16.93
m_6	1.0	0.0	0.0	-1.0	0.0	1.0	0.0	0.0	0.0	0.0	6.14
x	0.0	0.0	0.0	0.14	0.0	0.06	0.0	0.0	1.0	0.0	1.44
m_2	0.0	1.0	0.0	1.0	0.0	-1.0	0.0	0.0	0.0	0.0	2.63
y	0.0	0.0	0.0	-0.24	0.0	0.24	0.0	0.0	0.0	1.0	0.42
m_7	0.0	0.0	0.0	2.0	0.0	-1.0	1.0	0.0	0.0	0.0	18.69
m_8	0.0	0.0	0.0	-2.0	0.0	1.0	0.0	1.0	0.0	0.0	8.77

Cuadro 4.8: Último estado de la tabla. Aquí los valores para las celdas (c, x) y (c, y) vuelven a ser los mismos que en el inicio de la segunda fase del algoritmo, indicando que se terminó de recorrer todo el perímetro del polígono.

Conociendo las posiciones de los vértices del polígono que delimita la PZB de la red, es posible representarlo gráficamente en el espacio recíproco (figura 4.16). Analizando la complejidad de este algoritmo, se observa que el número de repeticiones de las operaciones necesarias para *meter* y *sacar* filas de la matriz, está limitado por el número de vértices del polígono resultante. Este número, debido a las propiedades de teselación del plano⁴, no puede exceder seis como consecuencia de restricciones geométricas [25].

Por lo tanto, la complejidad de nuestro algoritmo para determinar la PZB de una red es independiente de la red específica, ya que está acotada por una constante. Esto implica que pertenece a la clase de complejidad $O(1)$.

En la ejecución del software, al aplicar este algoritmo para un sistema, este se ejecuta un número de veces igual al numero de capas que conforman dicho sistema más uno (el correspondiente a la CP calculada para el sistema), por lo que su complejidad final

⁴Un polígono convexo puede teselar el plano si y solo si los ángulos interiores permiten que los polígonos encajen sin dejar huecos, de manera que la suma de los ángulos alrededor de un vértice sea igual a 360° .

pertenece a la clase $O(m)$ con m igual al número de capas del sistema.

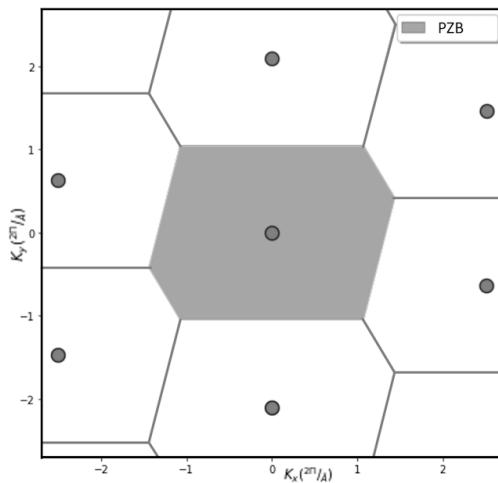


Figura 4.16: Imagen dada por el programa de la PZB de la red evaluada obtenida a partir del algoritmo descrito.

Funciones

B.1 Basics

B.1.1 Operaciones con vectores

En todo el código, se realizan operaciones con vectores. Dado que estamos trabajando con periodicidad en dos dimensiones, la mayoría de los vectores utilizados son bidimensionales. En este programa, representamos estos vectores mediante pares de números que indican la posición en el plano del punto correspondiente al vector que estamos representando (su tipo sería descrito como `v2d::(float,float)`).

En el caso de las operaciones de producto punto y producto cruz, operamos con vectores tridimensionales. En estos casos, un vector se representa con un conjunto de tres números que indican la posición en el espacio del punto que representa este vector (en este caso su tipo sería descrito como `v3d::(float,float,float)`).

A continuación, se presentan las funciones que realizan operaciones sobre vectores:

- **sumaV(\vec{a}, \vec{b})**:: `v2d,v2d → v2d`.
Suma los vectores \vec{a} y \vec{b} .
- **multV(n, \vec{a})**:: `v2d,v2d → v2d`.
Multiplica el vector \vec{a} por la constante n .
- **m2V(\vec{a}, \vec{b}, s)**:: `v2d,v2d,(float,float) → v2d`.
Recibe los vectores \vec{a} , \vec{b} y un par de números $s = (m, n)$. Calcula la combinación lineal $m\vec{a} + n\vec{b}$.
- **rota(\vec{v}, θ)**:: `v2d,float → v2d`.
Regresa el vector resultante de rotar el vector \vec{v} por un ángulo de θ grados.
- **dist(\vec{a}, \vec{b})**:: `v2d,v2d → float`.
Calcula la distancia entre los puntos representados por los vectores \vec{a} y \vec{b} .

- **long(\vec{v})**:: v2d → float.
Calcula la magnitud del vector \vec{v} .
- **cRot(\vec{v})**:: v2d → float.
Devuelve la dirección del vector \vec{v} en grados.
- **cAng(\vec{a}, \vec{b})**:: v2d, v2d → float.
Calcula el ángulo entre los vectores \vec{a} y \vec{b} , el resultado toma en cuenta la dirección del ángulo, por lo que puede ser positivo (si este es en sentido antihorario) o negativo (si es en sentido horario).
- **pC(\vec{a}, \vec{b})**:: v3d, v3d → v3d.
Calcula el producto cruz de los vectores \vec{a} y \vec{b} .
- **pP(\vec{a}, \vec{b})**:: v3d, v3d → v3d.
Calcula el producto punto de los vectores \vec{a} y \vec{b} .
- **to2D(\vec{v})**:: v3d → v2d.
Obtiene la proyección bidimensional del vector tridimensional \vec{v} .
- **to3D(\vec{v})**:: v2d → v3d.
Obtiene un vector 3D equivalente al vector 2D dado \vec{v} .

B.1.2 Operaciones con matrices

En el programa Trabaja solamente con operaciones en matrices de 2×2 , estas son representadas por una lista de 2 listas de 2 números que representarán sus filas ($m2x2::[[float, float], [float, float]]$).

Las operaciones que realizan operaciones sobre las matrices son las siguientes:

- **sumaM(A, B)**:: m2x2, m2x2 → m2x2.
Suma las matrices A y B .
- **multM(c, M)**:: float, m2x2 → m2x2.
Multiplica la matriz M por la constante c
- **VtM(\vec{u}, \vec{v})**: Genera una matriz $2x2$ a partir de dos vectores 2D dados usándolos como columnas de esta.
- **MtV(M)**:: m2x2 → v2d, v2d.
Convierte las 2 columnas de la matriz dada en vectores bidimensionales.
- **det(M)**:: m2x2 → float.
Calcula el determinante de la matriz M

- **inv2x2(M)**:: $m2x2 \rightarrow m2x2$.
Calcula la matriz inversa de la matriz M .
- **m2M(A, B)**:: $m2x2, m2x2 \rightarrow m2x2$.
Multiplica la matriz A por la matriz B .
- **transfVs(\vec{u}, \vec{v}, T)**:: $v2d, v2d, m2x2 \rightarrow v2d, v2d$.
Regresa como 2 vectores bidimensionales la matriz resultante de multiplicar la matriz $[\vec{u}, \vec{v}]$ por la matriz T .

B.1.3 Funciones auxiliares

A continuación se enlistan las funciones en Basics.py usadas como funciones auxiliares en funciones próximas.

- **getLim(\vec{u}, \vec{v}, m, n)**:: $v2d, v2d, int, int \rightarrow [float, float], [float, float]$.
Calcula los valores máximos y mínimos en X y Y del rectángulo donde está inscrito el paralelogramo formado por los vectores $m\vec{u}, n\vec{v}$. Es utilizada para determinar los límites del espacio para dibujar en pantalla.
- **esRotacion($\vec{a}, \vec{b}, \vec{c}, \vec{d}, \epsilon$)**:: $v2d, v2d, v2d, v2d, float \rightarrow bool$.
Determina si el par de vectores \vec{a}, \vec{b} son resultado de rotar el par de vectores \vec{c}, \vec{d} permitiendo un error máximo dado por ϵ , si este no se especifica se toma por defecto $\epsilon = 0,001$.
- **acomoda($x, valor, lis, tamMax$)**:: $object, float, [[object, float]], float \rightarrow [[object, float]]$.
Agrega el objeto x en la lista ordenada lis de acuerdo a su valor asociado $valor$, limitando el tamaño de la lista a $tamMax$.
- **pmat(M)**:: $m2x2 \rightarrow$.
Imprime una matriz M en pantalla como un String.
- **checkP($x, y, err = 0,001$)**:: $float, float, float \rightarrow bool$.
Evalúa si dos valores representan al mismo en un periodo de tamaño 1.
- **cRecip($\vec{a}, \vec{b}, \vec{c}$)**:: $v3d, v3d, v3d \rightarrow [v3d, v3d, v3d]$.
Dada una terna de Vectores tridimensionales, calcula sus respectivos Vectores recíprocos.
- **buscaEsquina($M, j, e1, e2$)**:: $[[float]], int, int, int \rightarrow int$.
Función auxiliar usada para calcular los vértices de la FBZ.

- **opera(M, e, s):** $[[\text{float}]], \text{int}, \text{int} \rightarrow$.

Usando operaciones en la Matriz M la transforma en otra donde la columna e sea de ceros excepto en la celda $M_{s,e}$ donde tiene el valor 1.

- **dameVecinos(rv):** $[\text{v3d}, \text{v3d}, \text{v3d}] \rightarrow [(\text{float}, \text{float})]$.

Regresa los 8 puntos de red más cercanos al origen dados por una red con los vectores primitivos señalados en rv .

- **califica($pos, lista$):** $(\text{float}, \text{float}), [(\text{float}, \text{float})] \rightarrow \text{bool}$.

Funcion auxiliar de *dameVecinos*, determina si un punto en la posición relativa pos no es “cubierto” por algún punto con una posición en $lista$ desde el origen.

- **calcVerticesFBZ(rv):** $[\text{v3d}, \text{v3d}, \text{v3d}] \rightarrow [(\text{float}, \text{float})], [[\text{float}]]$.

Calcula los vértices de la celda de Wigner-Seitz para la red con los vectores primitivos señalados en rv .

- **aN(a):** str->float

Analiza la entrada ‘ a ’, si esta coincide con el *símbolo atómico* de un elemento químico entonces regresa el *Número atómico* de este elemento, de lo contrario regresa 0,5

- **F_G(a, G):** (str, v2d)->float

Calcula el *factor de forma atómica* para el átomo con símbolo químico ‘ a ’ en la posición dada por el vector G ¹.

¹Los datos para el cálculo provienen de las Tablas Internacionales de Cristalográfia
<http://it.iucr.org/Cb/ch6o1v0001/>

B.2 Atom

B.2.1 Inicialización

El objeto *Atom* recibe como parámetros de inicio:

- **pos:: (float, float).**

Indica la posición relativa del Átomo a los vectores primitivos \vec{a} y \vec{b} de la red donde se encuentra.

- **posZ:: float.**

Indica la posición relativa del Átomo al vector primitivo \vec{c} de la red donde se encuentra. Si no se indica se toma el valor 0,0 por default.

- **color:: str.**

Indica el color con el que se pintará el Átomo al dibujarse en pantalla. Si no se indica se toma por default 'black'.

- **sig:: str.**

Identificador para señalar el tipo de Átomo al que pertenece este, se recomienda usar el Símbolo del Átomo. Si este no se indica se toma como valor por default 'C'.

Estos parámetros originan los Atributos homónimos del objeto Átom.

B.2.2 Métodos de Atom

Los métodos para Atom son los siguientes:

- **__str__0:: →str.**

Expresa Atom como un str de la forma: Atom.sig+str(Atom.pos).

- **setData(color,sig):: color,str→.**

Cambia los atributos 'color' y 'sig' del Atom.

- **getPos():: →(float, float).**

Regresa el atributo 'pos'.

- **clasifica(loa):: [[Atom]] →int.**

Dada *loa*, una lista de listas de Atoms clasificados por su atributo 'sig'. Si en *loa* hay una lista de objetos Atom con el mismo 'sig' que el Atom lo agrega a esta lista y regresa 1, de lo contrario genera una nueva lista en *loa* que lo contenga y regresa 0.

B.3 Lattice

B.3.1 Inicialización

Los objetos de clase Lattice modelan redes cristalinas, los parámetros de entrada para inicializar esta clase son:

- **vA:: v2d.**
Vector primitivo a de la red cristalina.
- **vB:: v2d.**
Vector primitivo b de la red cristalina.
- **atm:: [Atom].**
BA de la red cristalina.
- **enls:: [((float,float),(float,float))].**
Lista de las posiciones inicio y final de los segmentos de línea que se pintarán para representar los enlaces atómicos. (Solo requerido para representarse en pantalla dentro del programa, si este no se da o como parámetro se da una lista vacía, en las imágenes no se pintarán los enlaces pero no afecta en el archivo POSCAR generado al exportarse la Red o Sistema del que forme parte).
- **name:: str.**
Nombre que se le da a la red cristalina.
- **detachment :: float.**
Indica el grosor de la red cristalina en Angstroms.
- **prof:: int.**
Indica la capa a la que pertenece esta red si es que pertenece a un sistema apilado, si no es parte de un Sistema su valor es 1.

Los Atributos propios de esta Clase generados a partir de los parámetros de inicio son:

- **a:: v2d ; b:: v2d.**
Atributos que señalan los vectores primitivos \vec{a} y \vec{b} de la red, son dados directamente por vA y vB.
- **OriginalA:: v2d ; OriginalB:: v2d.**
Atributos que señalan los vectores primitivos \vec{a} y \vec{b} de la red, aquí se toman los vectores de la red rotada de tal manera que la dirección del vector \vec{a} sea de 0° .

- `prof:: int; detachment:: float; name:: str; enls:: [(float, float), (float, float)]`.
Atributos donde se guardan directamente los parámetros homónimos.
- `theta:: float ; inAngle:: float`.
Atributos que señalan ángulos de la Red, el primero es dado por la rotación requerida para llegar a la red por modelar a partir de la misma red si la dirección del vector \vec{a} fuera de 0° . El segundo ángulo es el ángulo inscrito entre los vectores \vec{a} y \vec{b} .
- `atms:: [[Atom]]`.
Lista de Listas de Atoms clasificados por su Atributo 'sig', esta es obtenida a partir del parámetro homónimo.
- `reciprocalVectors:: [v3d, v3d, v3d]`.
Terna de los vectores primitivos de la red reciproca correspondiente.
- `layerList :: [Lattice]`.
Si esta Lattice modela una Red cristalina formada por el apilamiento de 2 o más redes, aquí se enlistan las Lattices correspondientes a estas.

B.3.2 Métodos de Lattice

- `__str__0:: → str`.
Expresa el Lattice como un str con el formato de un archivo POSCAR.
- `get_pv:: → v2d, v2d`.
Regresa los vectores primitivos \vec{a} y \vec{b} .
- `showme(x, y, x0, y0, t, iName, sampling, scalePosL)::
int, int, int, int, float, str, bool, bool →`
Despliega en pantalla una representación de la super-red dada por los valores de x y y . Los parámetros $x0$ y $y0$ son usados si se requiere imprimir la super-red a partir de una posición distinta al origen. El parámetro t indica el grosor con el que se dibujarán los átomos y enlaces. El parámetro $iName$ indica el nombre del archivo png con el que se guardará la imagen generada (en la carpeta Images), si este parámetro es proporcionado, la imagen no se guardará y solo se desplegará en pantalla. Los parámetros booleanos `sampling` y `scalePosL` determinan correspondientemente si se dibujarán los vectores primitivos \vec{a}, \vec{b} en la imagen y si la posición de la escala se posicionará o no a la izquierda.
- `showPC(iName)::str→none`
Este llama al método `showme` con los parámetros necesarios para imprimir una imagen con solo la celda primitiva de la red indicando sus vectores primitivos. Igualmente que el método anterior, el parámetro $iName$ es opcional, si este es dado será

el nombre con el que se guardará la imagen, en caso de no darse la imagen solo se desplegará en pantalla sin guardarse.

- **addAtms(*loa*):: [Atom] →.**

Agrega de forma ordenada los Atoms en la lista *loa* al parámetro *atoms*.

- **showData()::→str**

Genera un texto con formato POSCAR con los datos de la red.

- **aligned()**

Rota la red de tal manera que el vector \vec{a} esté alineado al eje X.

- **rotate(θ):: float →.**

Rota la red en θ^o .

- **alignedLattice():: → Lattice.**

Regresa una copia de la red alineada al eje X.

- **mRot(θ):: θ → Lattice.**

Regresa una copia de la red rotada en θ^o .

- **export(*name*):: str →.**

Exporta la red en un archivo POSCAR <*name*>.vasp

- **setNewVectors(*newA*, *newB*):: v2d, v2d →.**

Establece nuevos valores para los atributos de Lattice *a* y *b*.

- **getVectors():: → v2d, v2d.**

Regresa los atributos de Lattice *a* y *b*.

- **getOV():: → v2d, v2d.**

Regresa los atributos de Lattice *OriginalA* y *OriginalB*.

- **nOAtms():: → int.**

Regresa el número de Átomos en la BA de la red.

- **loAtms():: → [[str, float, float, float, v2d]].**

Esta es una función auxiliar utilizada para calcular el patrón de difracción de la red. Genera una lista con datos de cada objeto Atom en la BA de la red (el atributo *atms* de la clase *Lattice*). Cada elemento de la lista de salida consta de los siguientes datos tomados de cada átomo: atributo *sig*, posición relativa X, posición relativa Y, posición relativa Z, vector 2D con la posición absoluta de la proyección del átomo en el plano.

- **F_hkl(h, k, FG)::float, float, bool → float**

Esta es una función auxiliar utilizada para calcular el patrón de difracción de la red. Calcula el Factor de estructura correspondiente al punto de la red reciproca h, k, l correspondiente (dado que nuestra estructura es 2D solo se utilizan h y k).

- **reciprocalBackgroundMesh($vl, t, border, calcS, rnd, FG$)::**

```
[(float, float)], float, float, bool, int, bool →
[float], [float], [float], linkList.
```

Esta es una función auxiliar utilizada para calcular el patrón de difracción y la representación del espacio reciproco de la red. Calcula los datos necesarios para dibujar es espacio reciproco de la red. La lista vl corresponde a los vértices de la FBZ de la red, el espacio que se muestra es limitado por $border$, rnd indica el número máximo de dígitos después del punto para redondear los resultados de la lista S que es parte de la salida de esta función. Los booleanos $calcS$ y FG determinan respectivamente si se calculará el factor de estructura (que será guardado en la lista S) y si para este cálculo se tomará en cuenta el vector de dispersión.

La salida de este método está formada por tres listas de valores reales y una lista de vértices, las primeras 2 listas (X, Y) guardan las posiciones de los puntos de la red reciproca dentro del espacio determinado por $border$, la tercera lista (S) tiene los valores del Factor de estructura calculados para cada punto (es vacía si $calcS = False$) y la lista de vértices tiene los vértices requeridos para crear una maya compuesta por calcular la FBZ en cada punto.

- **printReciprocalSpace($t, border, prnt, zoom, colors$)::**

```
float, float, bool, bool, [str] →.
```

Despliega en pantalla una representación del espacio reciproco de la red. Si el Lattice corresponde a una estructura de redes apiladas dibuja la FBZ de cada red que lo forma y la de esta misma. Los valores t y $border$ determinan el grosor general con el que se dibujará todo y el límite del espacio de dibujo respectivamente. Si el booleano $prnt = True$ entonces guarda la imagen en la carpeta “image”, si el $zoom = True$ entonces la imagen desplegada corresponderá solamente al primer cuadrante del plano. La lista $colors$ determina los colores con los que se dibujarán las FBZ de cada red en el sistema, si esta se proporciona debe indicar un color para cada capa del sistema, si no se proporciona entonces el color para cada capa será al azar.

- **printLightPoints($t, border, prnt, rnd$)::float, float, bool, int → none**

Despliega en pantalla una representación del patrón de difracción de la red. Los valores t y $border$ determinan el grosor general con el que se dibujará todo y el límite del espacio de dibujo respectivamente, rnd indica el número de dígitos después del punto con el que se redondeará el Factor de estructura de cada punto. Si $prnt = True$ se guardará la imagen en la carpeta “image”.

B.4 Functions

B.4.1 Funciones sobre átomos en una Base Atómica

- **isitin($r, cent, sr, slvl$)**:: Lattice, v2d, Lattice, int →.
Verifica qué átomos pertenecientes a la celda de la red r ubicada en la posición $cent$, caen dentro del área correspondiente a la CP de la red sr , los átomos que cumplen esto son agregados a la BA de sr .
- **megeCut(r, sr, lvl)**:: Lattice, Lattice, int →.
Identifica qué celdas de la red r intersectan la CP de sr y aplica sobre estos la función *isitin*.
- **cleanA(r, err)** :: Lattice, float →.
Verifica los Átomos pertenecientes a la BA de la red r y elimina las repeticiones. La variable err indica un límite de error aceptable, si este no se proporciona se toma 0,001 por default.
- **esClon($Atms, atm, \epsilon$)**:: [Atom], Atom, float → bool
Determina si el Átomo atm es equivalente a algún elemento de la lista $Atms$. Esta es una función auxiliar para eliminar posibles átomos repetidos en la BA de alguna red.
- **borders($Atms$)**:: [Atom] → [Atom]
Dada una lista de átomos correspondiente a la BA de un red, regresa una sublista con todos aquellos que son cercanos al borde de la CP de dicha red.
- **cleanA($Atms, \epsilon$)**:: [Atom], float → [Atom]
Evalúa una lista de átomos identificando aquellos que son equivalentes por periodicidad con un error delimitado por ϵ . Todos aquellos que identifica como clon "de alguno ya existente lo quita de la lista original y guarda en una lista nueva que es regresada por esta función.
- **cleanPCell(L, acc)**:: Lattice, int → [Atom]
Ejecuta la función *cleanA* sobre la báse atómica de la red L con un $\epsilon = 1/10^{\{acc\}}$.

B.4.2 Cálculo de la Celda Primitiva de una Red

En algunas de estas funciones se hace uso de una lista con formato específico para evaluar los PR, pList::[([(int, int)], v2d)], esta es una lista de elementos en pares donde la primera parte se señala el índice de la PR señalada y su vector adjunto, en la segunda parte se señala el *error* calculado para este PR.

- **pts(l, max_it)**:: Lattice, int → pList.
Crea una lista con los PR de l contenidos en la supercelda delimitada por max_it . El resultado se regresa en una lista pts con el formato $pList$.
- **calcCD(s, l, ab)**:: Lattice, Lattice, (int, int) → (int, int).
Calcula los índices c, d asociados al PR de la red l más cercano al PR de s asociado al índice a, b . Hace uso de la ecuación 4.2.
- **calcPR($pts, substrate, l, \epsilon$)**:: pList, Lattice, Lattice, float → pList.
Depura la lista pts eliminando aquellos cuyo PR en l asociado tiene un error superior al ϵ dado, si este valor de ϵ no se proporciona se toma 0,05 por default.
- **commonVs(Ls, max_val, ϵ)**:: [Lattice], int, float → [(int, int), float]
Usando cómo red ‘sustrato’ a la red en la posición 0 de la lista Ls , ejecuta secuencialmente la función calcPR con cada una de las demás redes en Ls , usando como lista pts inicial el resultado de ejecutar la función ‘ $pts(sustrato, max_val)$ ’. Quedando al final una lista con solamente los PRs que tienen un PR correspondiente en cada capa del sistema con un error inferior a ϵ .
- **corresponding_points($l1, l2, M1$)**:: Lattice, Lattice, m2x2 → m2x2
Regresa la matriz 2×2 que hace referencia al PR para $l2$ correspondiente al PR de $l1$ al que referencia $M1$.
- **calc_dd(Mi, Mo)**:: m2x2, m2x2 → float
Calcula el índice de deformación correspondiente a Mi si esta es deformada hasta ser igual a Mo .

B.4.3 Manejo de datos de Matrices de Trasformación

Los datos de las MTs por mostrar son tomados desde una lista de datos con formato $Mdata ::= [m2x2, m2x2, float, float, float]$, donde los datos corresponden en orden: la primer matriz corresponde a la MT de la que se va a mostrar información, la segunda es la matriz de deformación asociada a esta MT, los siguientes datos corresponden a la deformación que la matriz de deformación efectúa sobre los VPs de la red, los primeros dos al cambio de su magnitud y los últimos dos al cambio en su dirección.

- **textLonN(t, n, al)**:: str, int, str → str
Regresa un String con una longitud n cortando o agregando espacios al String t dado. La variable al indicará el alineamiento del String resultante en caso de agregar espacios, si $al = l$ los espacios se agregarán a la derecha del texto original, si $al = r$ los espacios se agregarán a la izquierda, si $al = c$ los espacios se agregarán equitativamente a la izquierda y derecha, si no se especifica un valor para al se tendrá por default que $al = c$.

- **header()::→str**
Genera el String que es usado como encabezado de las tablas con las que se desplegaran los datos de las MTs.
- **infLayer($L, data$)::Lattice, Mdata→str, int**
Genera un String desplegando la información contenida en $data$ y usando la información de la red L . Regresa el string generado y el número de átomos de la supercelda de L correspondiente a la MT asociado al $data$ dado.

B.4.4 Exportación de una Red desde archivo POSCAR

- **readFile($name$)::str→[str]**
Lee el archivo con nombre $name$ (la raíz de la dirección es en la que está código) y tratándolo como un archivo de texto plano transforma su contenido en una lista usando cómo elementos de esta cada línea de texto.
- **leeNumeros(s)::str→[float]**
Evalúa el texto s y regresa una lista con todos los números contenidos en este.
- **importLattice($name, prnt$)::str, bool→Lattice**
Crea un objeto Lattice a partir del archivo $name$ de formato POSCAR. Si $prnt = True$ se imprime en pantalla un mensaje indicando la creación del objeto Lattice y el nombre del archivo a partir del cual fue creado.

B.4.5 Redes prediseñadas

Estas funciones generan objetos Lattice prediseñados para redes hexagonales, rectangulares y para algunas redes cristalinas comunes.

- **hexa6($p, atms, name$)::float, [Atom], str→Lattice**
Genera una Red hexagonal con 6 simetrías radiales teniendo como constante de red p , $atms$ como su base atómica y por nombre $name$. La lista $atms$ es opcional, si esta no es dada entonces la red tendrá 2 átomos dentro de su base.
- **hexa3($p, atms, name$)::float, [Atom], str→Lattice**
Genera una Red hexagonal con 3 simetrías radiales teniendo como constante de red p , $atms$ como su base atómica y por nombre $name$. La lista $atms$ es opcional, si esta no es dada entonces la red tendrá 2 átomos dentro de su base.
- **rectLattice($p1, p2, atms, name$)::float, float, [Atom], str→Lattice**
Genera una Red rectangular con las constantes de red $p1$ y $p2$, los átomos señalados en la lista $atms$ como su BA y $name$ por nombre. La lista $atms$ es opcional, si esta no es proporcionada se generará la red con un solo átomo en el centro.

- **graphen()**::

Genera una red de Grafeno, con constante de red 2,44Å y con sus átomos centrados generando una red hexagonal con 6 simetrías radiales.

- **graphenC3()**::

Genera una red de Grafeno, con constante de red 2,44Å y con uno de sus átomos centrado en su base y otro en un vértice, generando una red hexagonal con 3 simetrías radiales.

- **blackPhosphorene()**::

Genera una red de Fosforeno Negro con las constantes de red 3,2601Å y 4,347Å.

- **h_BN()**::

Genera una Red que describe la face hexagonal del Nitruro de Boro laminar con constante de red de 2,512Å.

B.5 System

B.5.1 Inicialización

Los objetos de clase *System* modelan las estructuras (sistemas) formadas por apilamiento vertical de redes cristalinas bidimensionales, los **parámetros** de entrada para inicializar esta clase son:

- **lol::[Lattice]**

Lista de redes cristalinas apiladas que conforman el sistema. El orden de la lista determina el orden de apilamiento de las redes, así la red en la posición i de la lista estará inmediatamente sobre la red en la posición $i - 1$. La red en la posición 0 de la lista corresponderá a la capa inferior, esta será llamada capa sustrato.

- **name::str**

Nombre con el que se identificará al sistema.

Los **Atributos** propios para esta clase son:

- **redes::[Lattice]**

Lista de redes cristalinas apiladas que conforman el sistema.

- **name::str**

Nombre del sistema.

- **poits::[[int,int],float]**

Lista con todos los PR de la red en la capa sustrato, ligados a los VTs que son comunes para todas las redes del sistema. Al inicializar el objeto es una lista vacía.

- **loMat::[m2x2]**

Lista con las MTs que señalan una posible CP para el sistema. Al inicializar el objeto es una lista vacía.

- **MaxNumM::[int]**

Indica el número máximo de matrices guardados en la lista *loMat*. Al inicializar el objeto este atributo es igual a 10.

- **SuperRed::Lattice**

Guarda la Red que representa al sistema completo en base a la CP calculada. Al inicializar el objeto es “None”.

- **MT::m2x2**

Guarda la MT que señala la CP calculada. Al inicializar el objeto es “None”.

B.5.2 Métodos de System

Métodos principales

- **searchLP(*rangeOfSearch, epsilon*)**:: int, float → int

Calcula y guarda en el atributo *points* del sistema una lista con todos los PRs comunes para todas las redes del sistema en el área de búsqueda indicada por *rangeOfSearch* con un error menor a *epsilon*.

- **calculateTM(*min_angle*)**:: bool → int

Este método calcula las MTs que indican una CP para el sistema basados en los VTs indicados en la lista *poits* del sistema, las MTs calculadas se guardan en la lista *loMat*. Si la lista *poits* es vacía, muestra un mensaje de error y regresa -1. Si identifica que todas las redes del sistema son hexagonales, calcula solamente MTs que generan CPs hexagonales y regresa 0. Si **no** todas las redes del sistema son hexagonales regresa 1. La variable *min_angle* indica el ángulo interno mínimo que deben tener las CPs correspondientes a las MTs, por default es 20° .

- **createSuperLattice(*M*)**:: m2x2 → int

Crea y guarda en el atributo del sistema *SuperRed*, la red que represente al sistema completo usando la CP señalada por *M*. Si se creó esta red sin problema se regresa 1, de lo contrario se regresa 0. La red creada con este método puede tener imperfecciones dado que **no** deforma las redes que conforman al sistema.

- **optimize_system(*T, prnt*)**:: m2x2, bool → System, [m2x2]

Crea una copia del sistema y deforma las redes de este de forma que, con estas redes deformadas, no se genere imperfecciones al replicar de forma periódica la CP resultante del método *createSuperLattice* sobre este sistema copia deformado con la la MT *T*. Se copian los atributos *SuperRed* y *MT* del sistema copia al sistema original y se regresan el sistema deformado generado y una lista con las matrices de deformación usadas en cada capa de este.

- **ShowTMs(*shw, save*)**:: bool, str → m2x2

Efectúa un análisis sobre todas las MTs en *loMat* obteniendo y regresando aquella MT con un menor grado de distorsión (DD)². Si *shw = True* imprime en pantalla tablas con las características de las CPs ligadas a las MTs en *loMat*. Brindar la variable *save* es opcional, si esta es dada se creará un archivo de txt con el nombre dado en *save* donde estarán escritas las tablas evaluadas.

- **manualAdjustment(*TMs*)**:: [m2x2] → [m2x2]

Deforma las redes del sistema de manera que, al transformar las redes con las MTs

²Esta medida se calcula con la ecuación 4.10 en la sección Cálculo de Matriz de deformación del capítulo Implementación.

descritas en TMs , las superceldas resultantes tengan las mismas forma y dimensión entre si. Las MDs necesarias para esto son regresadas en una lista.

- **show()**:: none → none

Despliega en pantalla una imagen de la CP calculada para el sistema, una representación del la PZB de cada red del sistema y de la CP calculada, además de mostrar la MT usada para calcular la CP.

- **diffractionPattern($t, border, prnt$)**:: float, float, bool → none

Despliega en pantalla el Patrón de difracción del sistema a partir de la función *printLightPoits* sobre la red *SuperRed* del sistema con los parámetros $t, border$ y $prnt$ dados.

Métodos rápidos

Estos métodos ejecuta secuencialmente algunos métodos principales para abreviar instrucciones.

- **ejecuta(n, e)**:: int, float → m2x2

Efectúa secuencialmente las funciones *serchLP*, *calculateTM* y *ShowTMs*, sin desplegar mensajes en pantalla. Regresa la MT asociada a la CP pequeña con menor distorsión para el área de búsqueda señalada por n con error menor a e .

- **generateSuperCell(RoS, eps, sd)**:: int, float, bool → System

Efectúa de forma secuencial las funciones *serchLP*, *calculateTM*, *ShowTMs* y *optimize_system*, regresando un Sistema deformado con su CP calculada a partir de la MT obtenida por la función *ShowTMs*, la cual corresponde a la MT asociada a la CP pequeña con menor distorsión para el área de búsqueda señalada por *RoS* con error menor a *eps*.

Métodos auxiliares

- **clon()**:: none → System

Regresa un sistema copia.

- **set_long_loMat($newMax$)**:: int → none

Indica un nuevo número máximo de MTs guardadas en *loMat*.

- **analyze_T(T)**:: m2x2 → (m2x2, v2D, int, [Mdata])

Analiza la MT T y regresa los datos que requiere el método *leeMT* para crear la tabla correspondiente.

- **describeTM($T, prnt, shw$)**:: m2x2, str, bool → str, float

Crea una tabla con los datos correspondientes a la CP asociada a la MT T , regresa un str con este texto y la DD correspondiente al sistema con esta MT. La variable

prnt es opcional, si este se proporciona, el texto de la tabla es guardado en un archivo txt con ese nombre. Si *shw* = *True* la tabla calculada se despliega en pantalla.

- **its_hexagonal_system()**:: none → bool
Si todas las redes del sistema son redes hexagonales regresa *True*.
- **adjust(*M*)**:: m2x2 → int
Verifica si la MT dada *M* puede ser agregada a la lista *loMat*, si el determinante de la matriz es cero entonces regresa 0, de lo contrario regresa 1. Si *M* puede agregarse a *loMat* actualiza la lista.

Bibliografía

- [1] A. K. Geim and K. S. Novoselov, "The rise of graphene," *Nature materials*, vol. 6, no. 3, pp. 183–191, 2007.
- [2] Y. Liu, N. O. Weiss, X. Duan, H.-C. Cheng, Y. Huang, and X. Duan, "Van der waals heterostructures and devices," *Nature Reviews Materials*, vol. 1(9), p. 16042, 2016.
- [3] F. H. L. Koppens, T. Mueller, P. Avouris, A. C. Ferrari, M. S. Vitiello, and M. Polini, "Photo-detectors based on graphene, other two-dimensional materials and hybrid systems," *Nature Nanotechnology*, vol. 9, p. 780–793, 2014.
- [4] A. K. Geim and I. V. Grigorieva, "Van der waals heterostructures," *Nature*, vol. 499, no. 7459, pp. 419–425, 2013.
- [5] H. Ibach and H. Lüth, *Solid-state physics: an introduction to principles of materials science*. Springer Science & Business Media, 2009.
- [6] S. K. Dutta, S. K. Mehotor, and N. Pradhan, "Metal semiconductor heterostructures for photocatalytic conversion of light energy," *The Journal of Physical Chemistry Letters*, 2015.
- [7] G. Bastard, *Wave mechanics applied to semiconductor heterostructures*. Les Editions de Physique, 2016.
- [8] Z. Sun, Z. Yang, J. Zhou, M. H. Yeung, W. Ni, H. Wu, and J. Wang, "A general approach to the synthesis of gold-metal sulfide core-shell and heterostructures," *Angewandte Chemie (International ed. in English)*, vol. 48, no. 16, pp. 2881–2885, 2009.
- [9] Y. Ye, Z. J. Wong, X. Lu, X. Ni, H. Zhu, X. Chen, Y. Wang, and X. Zhang, "Monolayer excitonic laser," *Nature Photonics*, vol. 9(11), pp. 733–737, 2015.
- [10] F. Withers, O. Del Pozo-Zamudio, A. Mishchenko, A. P. Rooney, A. Gholinia, K. Watanabe, T. Taniguchi, S. J. Haigh, A. K. Geim, A. I. Tartakovskii, and K. S. Novoselov, "Light-emitting diodes by band-structure engineering in van der waals heterostructures. nature materials," *Nature Materials*, vol. 14(3), pp. 301–306, 2015.
- [11] G. Grosso and G. P. Parravicini, *Solid state physics*. Academic press, 2013.
- [12] Y. Liu, J. N. B. Rodrigues, Y. Z. Luo, L. Li, A. Carvalho, M. Yang, E. Laksono, J. Lu, Y. Bao, H. Xu, S. J. R. Tan, Z. Qiu, C. H. Sow, Y. P. Feng, A. H. C. Neto, S. Adam, J. Lu, and

- K. P. Loh, "Tailoring sample-wide pseudo-magnetic fields on a graphene–black phosphorus heterostructure," *Nature Nanotech*, vol. 13, pp. 828–834, 2018.
- [13] R. G. Parr and Y. Weitao, *Density-Functional Theory of Atoms and Molecules*. Oxford University Press, 1994.
- [14] P. Lazić, "Cellmatch: Combining two unit cells into a common supercell with minimal strain," *Computer Physics Communications*, vol. 197, pp. 324–334, 2015.
- [15] T. Necio and M. Birowska, "Supercell-core software: a useful tool to generate an optimal supercell for vertically stacked nanomaterials," *AIP Advances*, vol. 10, no. 10, 2020.
- [16] S. Naik, M. H. Naik, I. Maity, and M. Jain, "Twister: Construction and structural relaxation of commensurate moiré superlattices," *Computer Physics Communications*, vol. 271, no. 108184, 2022.
- [17] C. DiBona, S. Ockman, and M. Stone, *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, Inc., 1999.
- [18] S. Chazallet, *Python 3 los fundamentos del lenguaje*. Ediciones Eni, 2020.
- [19] G. E. Martin, *Transformation geometry: An introduction to symmetry*. Springer Science & Business Media, 2012.
- [20] B. Davvaz, *Groups and Symmetry: Theory and Applications*. Singapore: Springer Singapore, 2021.
- [21] G. Kresse and J. Furthmüller, "Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set," *Phys. Rev. B*, vol. 54, pp. 11169–11186, 1996.
- [22] K. Momma and F. Izumi, "VESTA3 for three-dimensional visualization of crystal, volumetric and morphology data," *Journal of Applied Crystallography*, vol. 44, no. 6, pp. 1272–1276, 2011.
- [23] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*. McGraw-Hill Education, 10 ed., 2014.
- [24] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. Wiley, 4th ed., 2010.
- [25] B. Grünbaum and G. C. Shephard, *Tilings and Patterns*. New York: Dover Publications, 2nd ed., 2016.