Universidade Federal da Paraíba
Centro de Informática
Programa de Pós-Graduação em Modelagem Matemática e Computacional

PYCRYSTALSCD: UM *SOFTWARE* PARA OBTENÇÃO DA DEMARCAÇÃO DO *CLUSTER* SUPRAMOLECULAR EM CRISTAIS ORGÂNICOS

Rodrigo Nóbrega Rocha Xavier

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Modelagem Matemática e Computacional, UFPB, da Universidade Federal da Paraíba, como parte dos requisitos necessários à obtenção do título de Mestre em Modelagem Matemática e Computacional.

Orientadores: Gerd Bruno da Rocha
Paulo Roberto dos Santos Salbego

João Pessoa
Dezembro de 2020

PYCRYSTALSCD: UM *SOFTWARE* PARA OBTENÇÃO DA DEMARCAÇÃO DO *CLUSTER* SUPRAMOLECULAR EM CRISTAIS ORGÂNICOS

Rodrigo Nóbrega Rocha Xavier

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL (PPGMMC) DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DA PARAÍBA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM MODELAGEM MATEMÁTICA E COMPUTACIONAL.

Examinada por:

Prof. Dr. Gerd Bruno da Rocha

Prof. Dr. Hugo Leonardo Davi de Souza Cavalvante

Jairo Rocha de Faria
SIAPE 1565613
Prof. Dr. Jairo Rocha de Faria

Prof. Dr. Paulo Roberto dos Santos Salbego

Prof. Dr. Sidney Ramos de Santana

JOÃO PESSOA, PB – BRASIL
DEZEMBRO DE 2020

X3p Xavier, Rodrigo Nóbrega Rocha
PyCrystalSCD: Um *software* para obtenção da demarcação do *cluster* supramolecular em cristais orgânicos / Rodrigo Nóbrega Rocha Xavier. – João Pessoa, 2020.
75 f.: il.;
Orientadores: Gerd Bruno da Rocha , Paulo Roberto dos Santos Salbego
Dissertação (mestrado) – UFPB/CI/PPGMMC.
Referências Bibliográficas: p. 44 – 47.
1. Cristalografia. 2. Clusters supramoleculares. 3. Diagrama de Voronoi. 4. Python.

UFPB/BC CDU: 548(043)

Ao meu filho Ravi e à minha esposa Rafaela, por suas presenças transformadoras em minha vida.

Agradecimentos

À minha esposa Rafaela, que tanto contribuiu com motivação e companheirismo, ajudando a me organizar e seguir em frente para a finalização deste trabalho.

Ao meu filho Ravi, cuja energia e alegria contagiante me apoiaram nos momentos difíceis.

Ao professor Gerd, pela orientação e ensino. A paciência e apoio sempre presentes e o amor ao ensino da química e de estratégias de programação foram fundamentais para o sucesso deste trabalho.

Ao professor Paulo Salbego, pela coorientação. Muito obrigado pelo repasse de conhecimento, apoio, troca de ideias, disponibilização de arquivos CIF e teste do resultado do *software*.

Ao professor Marcos Martins, pela pesquisa, desenvolvimento da ideia e repasse de conhecimento, e por viabilizar esta parceria entre UFSM e UFPB.

Aos professores Sidney, Jairo e Hugo, por aceitarem prontamente participar da banca e por darem contribuições valiosas para este trabalho final.

Ao professor Thiagão, que me despertou o gosto pela parte conceitual da Matemática Computacional com sua didática excepcional.

Aos demais professores que me ensinaram nesse mestrado: Ana, Claudio e Miguel. Obrigado pela disponibilidade e compreensão sempre que precisei.

A todos os colegas do mestrado em Matemática Computacional, em especial Neto, Ronally, Carlos, Wilson, Rafael, Dionarte e Manuel, sempre companheiros e dispostos a ajudar.

Aos colegas do LQQC.

Aos alunos do NUQUIMHE-UFSM que realizaram a atualização do POP 01 – a clareza da nova versão do documento ajudou a dirimir dúvidas e direcionar o trabalho. E aos que auxiliaram na repetição manual do procedimento para comparação da eficácia do *software* (Gustavo, João, Leandro, Darlon, Pedro, Priscila, Eduarda, Mariana, Gabrielle e Ramon).

Às instituições de fomento CAPES e CNPq e à UFPB, pelo financiamento do mestrado e disponibilização do laboratório LQQC.

E a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho.

Resumo da Dissertação apresentada ao PPGMMC/CI/UFPB como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PYCRYSTALSCD: UM *SOFTWARE* PARA OBTENÇÃO DA DEMARCAÇÃO DO *CLUSTER* SUPRAMOLECULAR EM CRISTAIS ORGÂNICOS

Rodrigo Nóbrega Rocha Xavier

Dezembro/2020

Orientadores: Gerd Bruno da Rocha
Paulo Roberto dos Santos Salbego

Programa: Modelagem Matemática e Computacional

O *cluster* supramolecular, segundo a abordagem proposta pelo grupo de pesquisa NUQUIMHE, é composto por uma molécula central (M1) e as MN moléculas do seu entorno, sendo a menor porção de uma estrutura cristalina capaz de trazer todas as informações topológicas e energéticas que caracterizam as interações intermoleculares envolvidas em toda a rede cristalina. A correta identificação do *cluster* supramolecular é o ponto de partida para a construção do entendimento da formação de determinada fase cristalina, o que possui extensa aplicabilidade. Porém, esse é um procedimento muito lento e sujeito a falhas, quando executado manualmente com apoio de programas existentes. Neste trabalho, é relatada a construção do *software* PyCrystalSCD, responsável por automatizar a parte mais trabalhosa dessa abordagem, que é a demarcação do *cluster* supramolecular através da obtenção da área de contato entre as moléculas envolvidas, utilizando diagramas 3D de Voronoi, além da geração de arquivos utilizados nas etapas posteriores da análise do cristal. O código final alcançou um rendimento médio 88 vezes mais rápido que o manual para executar um *dataset* de comparação.

Abstract of Dissertation presented to PPGMMC/CI/UFPB as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PYCRYSTALSCD: A SOFTWARE TO OBTAIN THE SUPRAMOLECULAR CLUSTER DEMARCATION IN ORGANIC CRYSTALS

Rodrigo Nóbrega Rocha Xavier

December/2020

Advisors: Gerd Bruno da Rocha
Paulo Roberto dos Santos Salbego

Program: Computational Mathematical Modelling

The supramolecular cluster, according to an approach proposed by the NUQUIMHE research group, is composed by a central molecule (M1) and the MN molecules around it, being the smallest portion of a crystal structure capable of providing all the topological and energetic information that characterize the intermolecular interactions involved throughout the entire crystal lattice. Correct identification of the supramolecular cluster is the starting point to understand the formation of a certain crystalline phase, which has extensive applicability. However, this is a very slow and fault-prone procedure when performed manually with the support of existing programs. In this work, the creation of the PyCrystalSCD software is reported, which is responsible for automating the most laborious part of that approach, which is the demarcation of the supramolecular cluster by obtaining the contact area between the involved molecules, using 3D Voronoi diagrams, in addition to generating files used in later stages of crystal analysis. The final code was 88 times faster than the manual procedure to perform a comparison dataset.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

$\boldsymbol{\Omega}$	Polígono genérico com vértices coplanares em um ambiente 3D, p. 31
$\boldsymbol{\alpha}$	Um dos ângulos dos parâmetros de rede de um cristal, medindo a inclinação entre as arestas b e c, p. 5
$\boldsymbol{\beta}$	Um dos ângulos dos parâmetros de rede de um cristal, medindo a inclinação entre as arestas a e c, p. 5
$\boldsymbol{\gamma}$	Um dos ângulos dos parâmetros de rede de um cristal, medindo a inclinação entre as arestas a e b, p. 5
a	Um dos parâmetros de rede de um cristal, medindo a distância entre a origem e o limite da cela unitária na direção do eixo x, p. 5
b	Um dos parâmetros de rede de um cristal, medindo a distância entre a origem e o limite da cela unitária na direção do eixo y, p. 5
c	Um dos parâmetros de rede de um cristal, medindo a distância entre a origem e o limite da cela unitária na direção do eixo z, p. 5
Å	Angstroms, unidade de medida utilizada no nível atômico, correspondente a 10^{-10} m, p. 26
$\mathscr{P}$	Plano que contém o polígono 3D, p. 31

Lista de Abreviaturas

CCDC	*Cambridge Crystallographic Data Centre*, p. 6
CIF	*Crystallographic Information File*, p. 6
CSDS	*Cambridge Structural Database System*, p. 19
CSD	*Cambridge Structural Database*, p. 1
CSV	*Comma Separated Value*, p. 34
DFT	*Density functional theory*, p. 12
IDE	*Integrated Development Environment*, p. 18
NUQUIMHE	Núcleo de Química de Heterociclos, p. 2
N	Número de coordenação molecular, p. 37
POP	Procedimento Operacional Padrão, p. 12
VDP	*Voronoi-Dirichlet Polyhedron*, p. 14

Capítulo 1

Introdução

Na área de engenharia de cristais, busca-se o emprego de estratégias ordenadas para o planejamento e construção (síntese) de materiais funcionais, em estado sólido, a partir de blocos de construção moleculares [1]. Isso é viabilizado através do entendimento e manipulação das interações intermoleculares, para que se possa produzir o cristal de interesse. Dessa forma, essa área de atuação emprega conceitos muito desenvolvidos de uma subárea da química conhecida como química supramolecular, que inclusive já rendeu um prêmio Nobel ao químico francês Jean-Marie Lehn em 1987, junto a Donald Cram e Charles Pedersen [2].

O entendimento de como produzir uma fase cristalina desejada tem impacto, por exemplo, no estudo das fases polimórficas de uma substância (que é quando a mesma substância existe em diferentes fases cristalinas), onde essas fases cristalinas distintas podem possuir propriedades diferentes e assim causar impactos importantes na indústria farmacêutica [3].

No início dos estudos envolvendo a engenharia de cristais, o desafio era o de produzir e determinar experimentalmente a estrutura de um cristal específico [4]. Hoje, essa barreira já foi superada pelos avanços da área e o desafio passou a ser o estudo, de forma sistemática, da crescente quantidade de estruturas de cristais disponíveis na literatura, principalmente com auxílio dos bancos de dados de estruturas cristalográficas (ex. CSD, o *Cambridge Structural Database* [5]). Neles, estão depositadas as estruturas cristalográficas de centenas de milhares de substâncias, incluindo compostos inorgânicos e orgânicos, possuindo polimorfismo ou não, dentre outros.

Nas pesquisas que são conduzidas hoje nesse campo, o interesse é o de promover uma análise conjunta da formação da estrutura supramolecular do cristal, levando-se em consideração não apenas seus detalhes geométricos, mas também topológicos e energéticos [6]. E é nesse ponto que a química teórica ganha um protagonismo importante, pois podemos aplicar diversas abordagens teóricas (principalmente as topológicas e energéticas) para contribuir substancialmente na: elucidação dos me-

canismos de formação dos cristais moleculares, no fenômeno do polimorfismo em cristais, entre outros [7].

Na literatura existem diversas estratégias teórico-experimentais como métodos para compreensão do cristal [4]. Mais recentemente, tomamos conhecimento da estratégia proposta pelo grupo de pesquisa NUQUIMHE (Núcleo de Química de Heterociclos, da Universidade Federal de Santa Maria) conhecida como método da demarcação do *cluster* supramolecular [8]. Nessa abordagem, o que se chama de *cluster* supramolecular é a parte do cristal que compreende o conjunto de MN moléculas da primeira esfera de coordenação no entorno de uma molécula central, a M1. Esse *cluster* supramolecular representa a menor porção de uma estrutura cristalina capaz de trazer todas as informações (topológicas e energéticas) acerca das interações intermoleculares envolvidas em toda a rede cristalina [8].

O procedimento para caracterizar o *cluster* supramolecular envolve a utilização de diversos programas específicos (ex. o Mercury [9] e o ToposPro [10]) que auxiliam no mapeamento de todas as moléculas que estão "em contato" com a M1 e participam dessa unidade básica. Uma vez caracterizada essa unidade básica, diversas análises podem ser conduzidas com o propósito de se entender como se deu a formação da sua estrutura cristalina. Em geral, essas análises são acompanhadas de cálculos sofisticados de química quântica.

Utilizando a demarcação do *cluster* supramolecular, o grupo de pesquisa NUQUIMHE já publicou uma série de estudos em que a validação dessa abordagem foi extensivamente aplicada. A demarcação foi explorada especialmente na compreensão dos mecanismos de cristalização de heterociclos[11][12][13][14], triazenos-N-óxidos[15] N-fenilamidas[16] e rotaxanos[17], avaliando o papel de diferentes interações intermoleculares. Outros estudos supramoleculares envolvendo essa demarcação foram realizados com líquidos iônicos dicatiônicos[18] e polimorfos[19][20]. Recentemente, o desafio voltou-se para o desenvolvimento de novos descritores para avaliar a similaridade supramolecular de estruturas cristalinas de cristais orgânicos[21][22].

O desafio de se aplicar esse procedimento é o fato de ele ser laborioso e demorado, limitando o estudo a poucas moléculas por vez. O processo envolve inspeção visual com auxílio dos dois programas já citados, bem como a tomada de decisão pelo usuário em várias situações conflitantes. Além disso, o usuário precisa armazenar diversas estruturas intermediárias, ou seja, todas as moléculas em contato com a M1 precisam ter a estrutura isolada para ser armazenada separadamente, repetindo o procedimento para cada possibilidade de escolha de M1. Mesmo tendo sido documentado um Procedimento Operacional Padrão (POP) de geração dos *clusters* supramoleculares, a possibilidade de se cometer erros em um protocolo manual é alta.

Assim, um *software* ou código que implemente o procedimento padrão de mon-

tagem de *clusters* supramoleculares de forma automática e segura torna-se urgente. Otimizando o tempo de coleta de dados e permitindo a análise de milhares de moléculas, seria possível competir em superioridade com os softwares atuais, que apresentam limitações no estudo sistemático de estruturas cristalinas. Ter uma ferramenta como essa possibilitaria propor teorias, a partir da observação de padrões, sobre a formação de cristais. Isso seria um importante auxílio na compreensão do processo de cristalização de polimorfos, solvatos, hidratos e cocristais.

Nesse sentido, os grupos de pesquisa do Prof. Martins (NUQUIMHE-DQ-UFSM) e do Prof. Gerd Rocha (DQ-UFPB) uniram esforços para propor uma ferramenta de automatização de montagem de *clusters* supramoleculares a partir de dados cristalográficos.

1.1 Objetivos

- Elaborar um *software* que obtenha a demarcação do *cluster* supramolecular a partir de arquivos com informações cristalográficas (.cif), utilizando diagramas 3D de Voronoi para calcular as áreas de contato entre as moléculas envolvidas.
- Testar a corretude do *software*, por meio da comparação entre seu resultado e o produzido por meio do protocolo manual existente.

1.2 Organização do Trabalho

O trabalho está organizado da seguinte forma:

Capítulo 2: Discorre sobre a fundamentação teórica relacionada com os objetivos do trabalho, começando com conceitos sobre cristalografia e explicações sobre como os cristais estão representados em arquivos CIF, apresentando em seguida o método do *cluster* supramolecular e os passos executados para aplicá-lo e finalizando com explicações sobre o poliedro de Voronoi-Dirichlet.

Capítulo 3: Informa o método proposto, detalhando as principais funcionalidades utilizadas com as bibliotecas encontradas e o funcionamento do algoritmo.

Capítulo 4: Informa os resultados obtidos e as melhorias previstas para o futuro.

Capítulo 2

Fundamentação Teórica

2.1 Cristalografia

Para alcançar os objetivos deste trabalho, um dos conceitos da química mais importantes a ser entendido é o conceito de cristal, já que o *software* que está sendo construído irá avaliar diferentes cristais a partir dos dados que definem cada um deles.

Um cristal é um sólido que assume um arranjo ordenado de átomos, íons ou moléculas. Tal arranjo é chamado de estrutura cristalina e é caracterizado por uma contínua repetição da mesma estrutura nas três dimensões do espaço, de forma perfeitamente regular. Assim, para todo cristal, é possível definir uma cela unitária, que é como um "bloco de construção" básico (na forma de um paralelepípedo) que, ao se repetir em todas as direções, forma toda a estrutura cristalina [23]. Como exemplo, a Figura 2.1 mostra a cela unitária do cristal com código CCDC WERPOY. Imagem obtida a partir do programa Mercury [9], após carregar o arquivo CIF que contém as informações desse cristal e selecionar a opção "*Packing*".

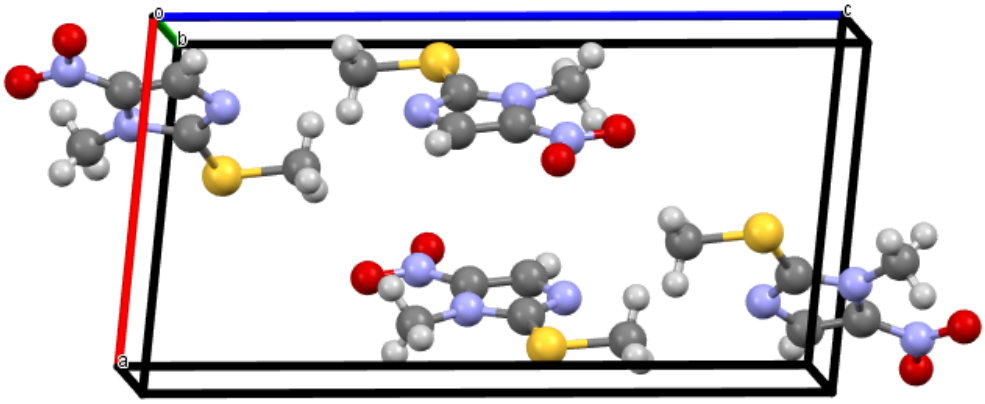

Figura 2.1: Cela unitária do cristal com código CCDC WERPOY.

Nesse exemplo, podemos visualizar os limites do paralelepípedo que define a cela unitária (em que **a**, **b** e **c** correspondem às medidas nos eixos x, y e z, respectivamente) e quais são as moléculas em que o centroide participa de seu interior (mesmo que parte da molécula esteja fora desses limites).

Para demonstrar como o cristal se comporta, a Figura 2.2 apresenta essa mesma estrutura cristalina após aumentar os limites nas opções de empacotamento do Mercury [9], com um aumento de 50%, em todas as direções (por exemplo, no eixo x, o limite em que o centroide deve estar situado para que a molécula seja exibida ficou sendo entre -0,5**a** e 1,5**a**, em vez de 0 a **a**). Mais moléculas são apresentadas, sempre repetindo a estrutura exatamente como disposto na cela unitária original.

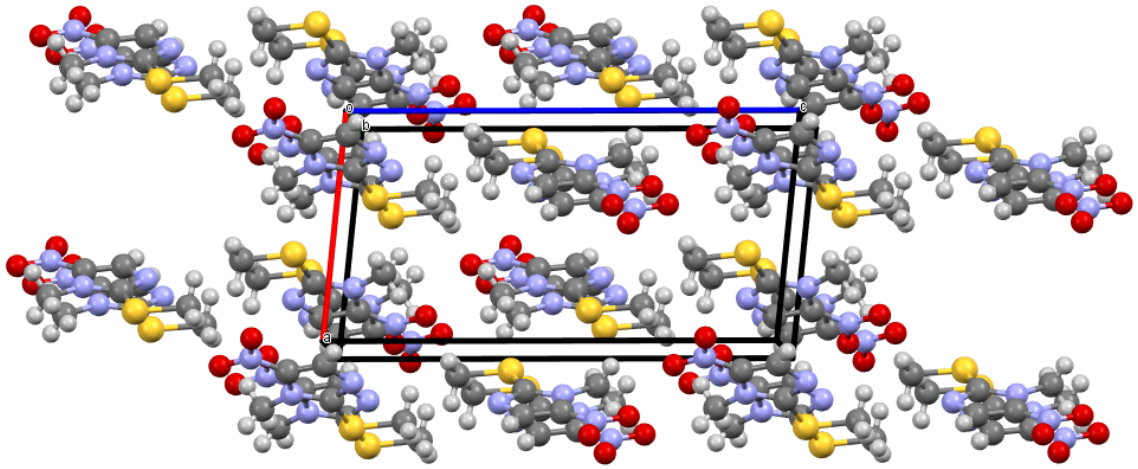

Figura 2.2: Expansão do cristal com código CCDC WERPOY.

Na definição da cela unitária, cada aresta pode ter diferentes tamanhos e os ângulos entre elas podem ser diferentes de 90°. Os três comprimentos (**a**, **b** e **c**) e os três ângulos ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ e $\boldsymbol{\gamma}$) que definem o formato da cela são chamados de parâmetros de rede e estão exemplificados na Figura 2.3.

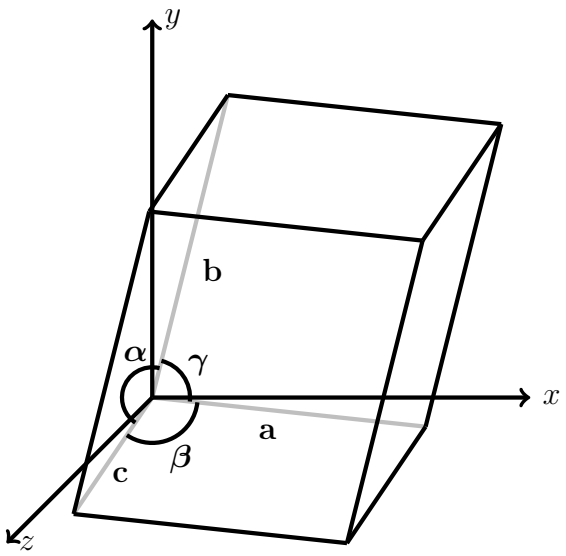

Figura 2.3: Cela unitária genérica e seus parâmetros de rede.

A forma cristalina que uma substância adota no estado sólido depende de como os átomos estão conectados, da sua geometria espacial, das interações intermoleculares e de parâmetros físico-químicos empregados durante o procedimento de cristalização (temperatura, pressão, solvente, pH, etc.) Uma vez formado o cristal, o posicionamento de cada átomo no espaço pode ser determinado através da técnica experimental de difração de raios X. O produto final que é gerado é um arquivo digital (formato .cif - *Crystallographic Information File*) em que todas as informações do empacotamento cristalino estão contidas (ou seja: todos os dados sobre a cela unitária). É esse arquivo que será usado como entrada para o *software* desenvolvido neste trabalho.

O arquivo CIF contém, entre outras informações, os seguintes dados sobre o cristal:

- Parâmetros de rede da cela unitária;
- Lista de átomos da(s) molécula(s) da unidade assimétrica e suas posições na cela unitária;
- Operadores de simetria.

Como exemplo, serão apresentados trechos do arquivo WERPOY.cif, obtido a partir do *Cambridge Crystallographic Data Centre* (CCDC), referente ao cristal com código CCDC WERPOY [24]. Os parâmetros de rede nesse arquivo estão listados da seguinte forma:

```
_cell_length_a 7.769(1)
_cell_length_b 6.575(1)
_cell_length_c 15.258(3)
_cell_angle_alpha 90
_cell_angle_beta 95.49(4)
_cell_angle_gamma 90
```

Os valores entre parênteses representam a margem de erro da última casa decimal. No exemplo acima, $\mathbf{a} = 7,769 \pm 0,001$ e $\boldsymbol{\beta} = 95,49 \pm 0,04$. Para os cálculos do *software*, no entanto, será sempre utilizado o valor médio nesses casos. O motivo de fazer essa escolha é que o programa utilizado durante o protocolo manual utiliza uma posição única para cada átomo para calcular as áreas de contato, que são somatórios de áreas de polígonos planos, como será explicado posteriormente. Assim, a busca por esse cálculo, que já vem sendo usado com sucesso, também tem que usar apenas os valores médios. Por isso, os valores entre parênteses serão ignorados.

A unidade assimétrica é a menor fração da cela unitária que, através da aplicação de operadores de simetria, gera a cela unitária completa [25]. Seguindo com o exemplo, as posições dos átomos da unidade assimétrica são assim definidas:

```
loop_
_atom_site_label
_atom_site_type_symbol
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
N1 N 0.7351(1) 0.4436(2) 1.0391(1)
C1 C 0.6811(3) 0.3077(3) 1.1071(1)
C2 C 0.7124(2) 0.4085(2) 0.9514(1)
S1 S 0.6076(1) 0.1919(1) 0.90871(3)
C3 C 0.6282(4) 0.2302(6) 0.7937(2)
N2 N 0.7753(2) 0.5567(2) 0.9044(1)
C4 C 0.8424(2) 0.6936(3) 0.9639(1)
C5 C 0.8191(2) 0.6287(3) 1.0469(1)
N3 N 0.8703(2) 0.7282(3) 1.1269(1)
O1 O 0.9422(2) 0.8936(3) 1.1230(1)
O2 O 0.8417(3) 0.6468(4) 1.1961(1)
H1 H 0.783(5) 0.273(5) 1.146(3)
H2 H 0.603(4) 0.369(6) 1.135(2)
H3 H 0.634(4) 0.192(6) 1.093(2)
H4 H 0.574(5) 0.120(6) 0.776(2)
H5 H 0.750(5) 0.229(5) 0.787(2)
```

```
H6 H 0.580(5) 0.366(8) 0.777(3)
H7 H 0.887(3) 0.805(4) 0.943(2)
```

Ao utilizar o comando `loop_`, significa que os campos determinados em seguida são repetidos por várias linhas, separados por um espaço entre seus valores. Assim, para o primeiro átomo listado, temos `_atom_site_label = N1`, `_atom_site_type_symbol = N`, `_atom_site_fract_x = 0,7351`, ... E na linha seguinte, temos `_atom_site_label = C1` e assim por diante. Observa-se que cada átomo distinto na unidade assimétrica recebe um rótulo (*label*) único, distinguindo assim os átomos do mesmo tipo. Os átomos da unidade assimétrica com seus rótulos e suas posições em relação à cela unitária estão representados na Figura 2.4 (imagem obtida a partir do Mercury [9]). Convenciona-se chamar de Z' o número de moléculas da unidade assimétrica. Neste exemplo, como a unidade assimétrica contém apenas uma molécula, então Z'=1.

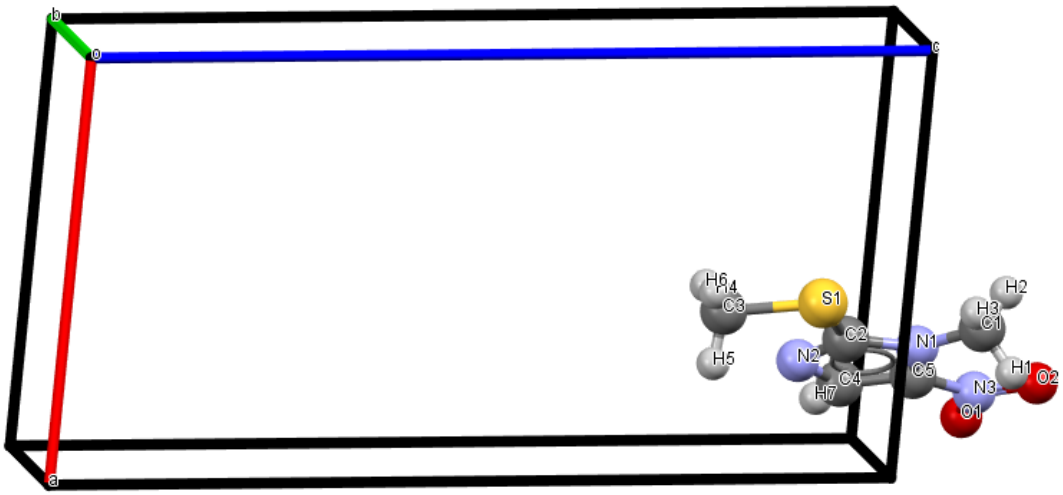

Figura 2.4: Átomos da unidade assimétrica do cristal com código CCDC WERPOY.

As posições dos átomos informadas no arquivo CIF são as coordenadas fracionárias, isto é, os valores relativos de cada coordenada variando de 0 a 1 no interior da cela unitária. Esse não é o valor da coordenada em angstroms, que é calculado multiplicando esse valor pelo comprimento correspondente da cela (por exemplo: se tivermos a coordenada fracionária x = 1, então a coordenada em angstroms tem o valor do parâmetro de rede **a**, e se a coordenada fracionária z = 0,5, então essa coordenada corresponde a metade do parâmetro de rede **c**). Por isso, no exemplo, alguns átomos têm `_atom_site_fract_z` > 1, o que os faz estarem situados fora da cela unitária. O mesmo aconteceria se alguma das coordenadas de um dos átomos fosse menor do que zero.

Quanto aos operadores de simetria, eles servem para informar quais são as operações matemáticas que são aplicadas à(s) molécula(s) da unidade assimétrica para obter as demais moléculas, permitindo executar operações de rotação, inversão, reflexão e translação. No arquivo CIF do exemplo, eles estão definidos da seguinte forma:

```
loop_
_symmetry_equiv_pos_site_id
_symmetry_equiv_pos_as_xyz
1  x,y,z
2  1/2-x,1/2+y,1/2-z
3  -x,-y,-z
4  -1/2+x,-1/2-y,-1/2+z
```

O primeiro operador de simetria (x,y,z) gera a própria molécula da unidade assimétrica. Os demais geram essa molécula em outras 3 posições e orientações diferentes. E, a partir dessas 4 moléculas, pode-se repetir com a mesma orientação e posição relativa em outras celas unitárias infinitas vezes (operação de translação). Para representar essa repetição, geramos um novo operador de simetria somando qualquer número inteiro (positivo ou negativo) a qualquer das coordenadas do operador de simetria original.

A Figura 2.5 (também obtida a partir do Mercury [9]) ajuda a esclarecer como essas operações de simetria funcionam. Ainda com o mesmo cristal, visto de um ângulo diferente, vemos a molécula da unidade assimétrica (operador x,y,z) e, na parte de baixo, o resultado da aplicação do operador -1/2+x,-1/2-y,-1/2+z. Soma-se 1 à coordenada y, gerando o operador -1/2+x,1/2-y,-1/2+z, que realiza a translação da molécula para a posição correspondente na cela vizinha.

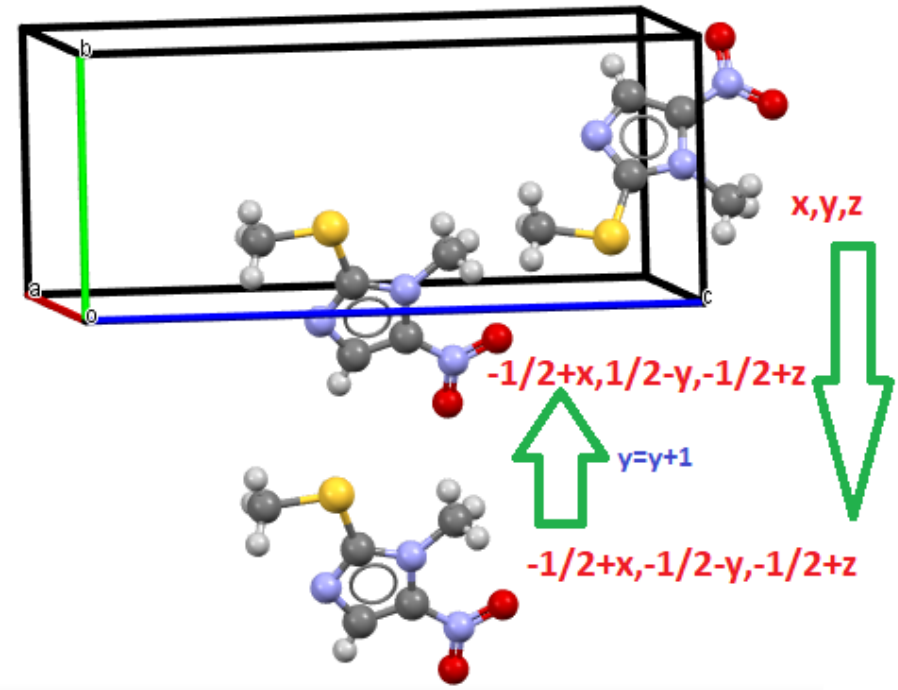

Figura 2.5: Exemplo de aplicação de operadores de simetria.

Quando esses operadores são aplicados, o efeito é a mudança de posição de todos os átomos para a nova posição definida pelo operador. Por exemplo: cada molécula desse cristal tem um único átomo de enxofre (S), representado na cor amarela. Na molécula da unidade assimétrica, as coordenadas fracionárias desse átomo são: (0.6076, 0.1919, 0.9087). Aplicando-se o operador -1/2+x,-1/2-y,-1/2+z, a posição desse átomo vai para (0.1076, -0.6919, 0.4087). E após somar 1 à coordenada y, a posição vai para (0.1076, 0.3081, 0.4087), por isso o operador -1/2+x,1/2-y,-1/2+z deixou esse átomo dentro da cela unitária original.

2.2 Método do *cluster* supramolecular

Para explicar como ocorre a formação do cristal, a estratégia proposta pelo grupo de pesquisa NUQUIMHE é o chamado método do *cluster* supramolecular [8]. Nesse método, é selecionada uma molécula central, denominada M1, e todas as moléculas em seu entorno que possuam interação com ela. Um exemplo pode ser visto na Figura 2.6, preparada no programa Mercury [9] e editada para identificar as moléculas. Esse conjunto de MN moléculas é chamado de *cluster* supramolecular e é suficiente para conter todas as informações topológicas e energéticas presentes em todo o cristal.

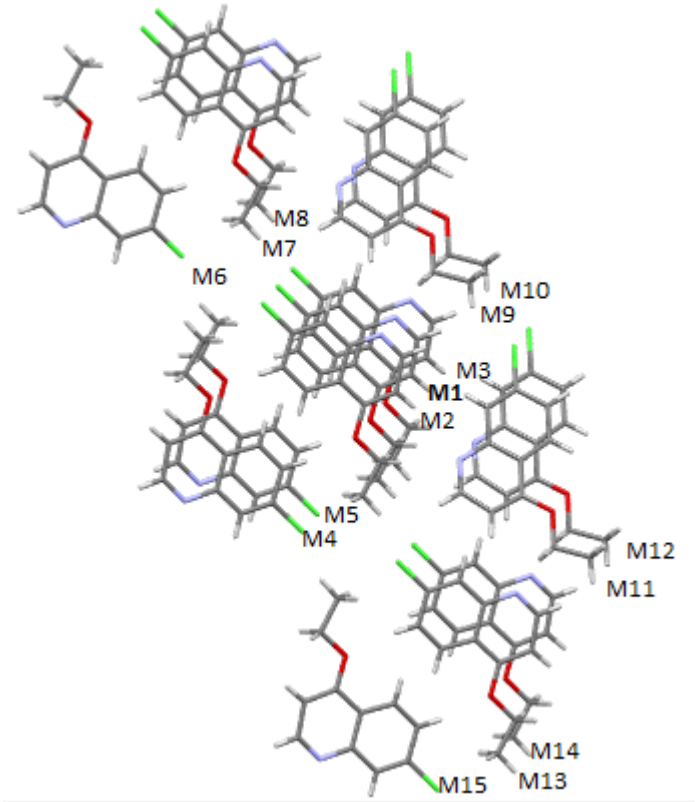

Figura 2.6: Exemplo de *cluster* supramolecular da estrutura com código CCDC LERKIE, contendo a M1 no centro e MN moléculas (de M2 a M15) no seu entorno.

Para identificar todas as MN moléculas do *cluster*, o primeiro passo é a escolha da M1. A escolha mais usual é a molécula da unidade assimétrica, quando única ($Z' = 1$), porém há casos em que esse operador contém várias moléculas ($Z' > 1$), sendo necessário selecionar uma ou um conjunto delas (chamando de M1 ao conjunto selecionado) e comparar os resultados ao final.

O segundo passo é expandir o cristal e construir um poliedro de Voronoi-Dirichlet que envolva a M1 completamente, obtendo a área de contato entre as demais moléculas e a M1 e salvando os dados delas. Isso garante que todas as moléculas que possuam interação com a M1 serão envolvidas nos cálculos a serem efetuados. O poliedro de Voronoi-Dirichlet tem sido construído com o auxílio do programa ToposPro [10], mas o trabalho de identificar a M1, selecionar as moléculas que formam o poliedro e verificar se ele envolve completamente a M1 são os passos manuais que mais tomam tempo no procedimento. Detalhes sobre o conceito e construção desse poliedro serão explicados posteriormente neste trabalho.

Em seguida, os dados das moléculas participantes no poliedro são utilizados para calcular a energia de estabilização dessas moléculas em relação à M1. Tais informações são obtidas por meio de cálculos de mecânica quântica no nível da Teoria funcional da densidade (*Density functional theory* - DFT), realizados pelo

programa Gaussian [26]. Enfim, a análise dos dados obtidos permite entender o mecanismo de cristalização das moléculas analisadas por meio da verificação das interações intermoleculares com maior ou menor energia de estabilização e área de contato.

As operações manuais realizadas pelo NUQUIMHE foram descritas de forma padronizada em documentos intitulados como POP (Procedimento Operacional Padrão) seguidos da numeração conforme a ordem dos procedimentos, dos quais iremos detalhar o primeiro deles [27].

2.2.1 Procedimento Operacional Padrão 01 - Obtenção de *cluster* supramolecular

O POP 01 [27] trata do procedimento que mais depende de tarefas manuais, que é a obtenção da demarcação do *cluster* supramolecular e das áreas de contato envolvidas, sendo essa a tarefa mais importante que o PyCrystalSCD conseguiu automatizar.

Inicialmente, o documento solicita que o arquivo .cif seja aberto com o programa Mercury [9]. Isso permite identificar a molécula da unidade assimétrica. Solicita-se também que seja habilitada a opção de exibir os eixos da cela unitária, para permitir localizar visualmente a molécula (o documento descreve os procedimentos para o caso de apenas uma molécula na unidade assimétrica, que é escolhida como a M1).

Em seguida, solicita-se que se abra o mesmo arquivo no programa ToposPro [10]. O programa exibe apenas os trechos das moléculas dentro da cela unitária, sendo necessário acionar uma opção que complete essas moléculas. Mas são mostradas também as moléculas que não pertencem à unidade assimétrica (geradas pelos operadores de simetria). Por isso, o próximo passo é comparar as moléculas do ToposPro [10] com a exibida no Mercury, até identificar com certeza que a M1 foi isolada no ToposPro e as demais omitidas.

Feito isso, é utilizada uma função que identifica os átomos com interação intermolecular com a molécula selecionada (a M1), fazendo então a expansão das moléculas correspondentes. O resultado contém, possivelmente, todas as moléculas do *cluster* supramolecular, o que será verificado em seguida.

Dando sequência, é necessário obter as áreas de contato entre a M1 e as demais moléculas encontradas e verificar se todo o entorno da M1 foi preenchido. Isso é feito através do poliedro de Voronoi-Dirichlet, que também é construído no ToposPro [10]. Cada face desse poliedro contém a área de contato entre um átomo da M1 e um átomo de outra molécula, que é somada para se obter a área de contato de cada molécula com a M1. Na Figura 2.7, vemos um exemplo de poliedro de Voronoi-Dirichlet gerado pelo ToposPro [10].

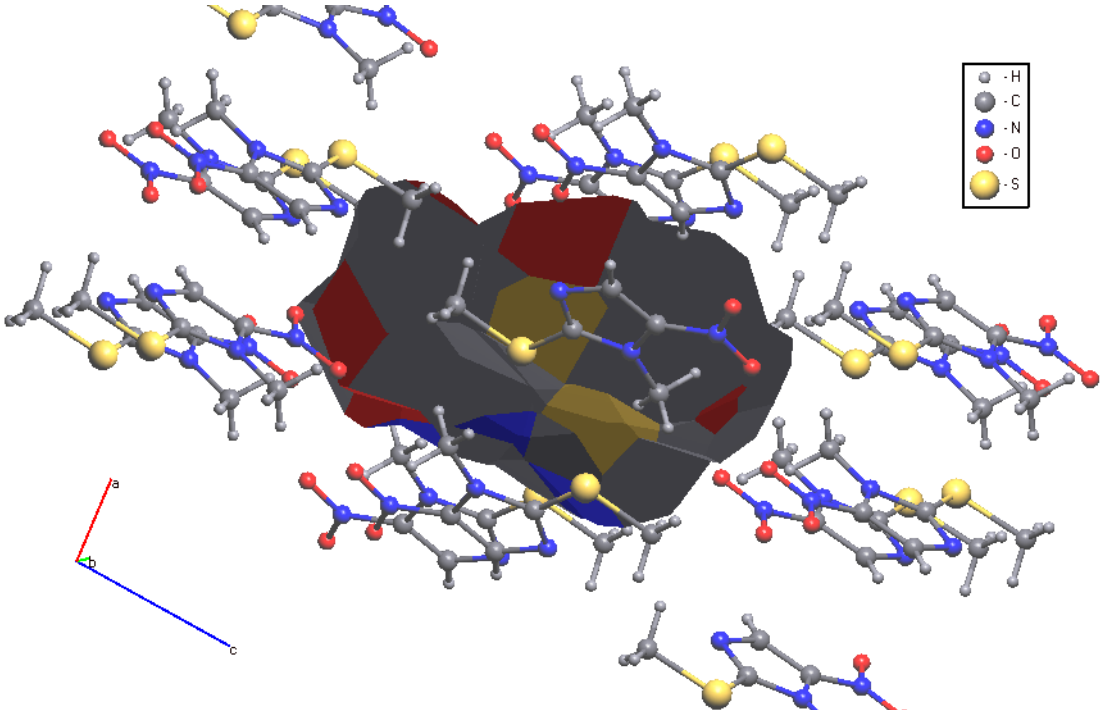

Figura 2.7: Poliedro de Voronoi-Dirichlet envolvendo completamente a M1 de um cristal (código CCDC: WERPOY).

Em seguida, é necessário verificar se há espaços abertos no poliedro, pois só há garantia de que o *cluster* supramolecular estará completo se o poliedro envolver a M1 completamente. Se forem encontrados buracos no poliedro, isso significa que falta selecionar moléculas no entorno da M1 e gerar novamente o poliedro. A Figura 2.8 mostra um caso em que isso acontece (imagem obtida do POP 01 [27], onde foi utilizada a estrutura de código CCDC LERKIE no programa ToposPro [10]).

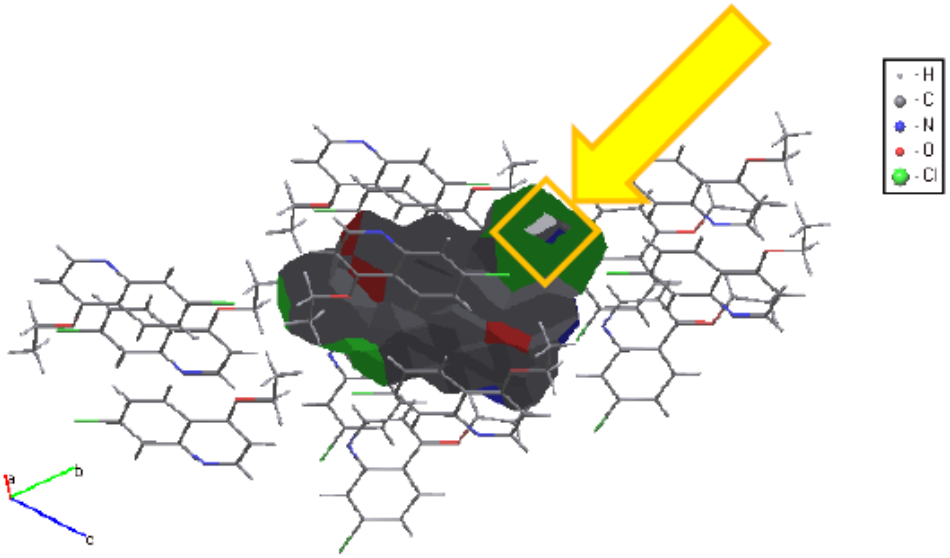

Figura 2.8: Poliedro de Voronoi-Dirichlet com um espaço faltando para completar.

Quando verificado que não há buracos no poliedro, podemos considerar que o *cluster* supramolecular foi identificado. Então, ainda é necessário identificar as MN moléculas uma a uma, o que é feito rotacionando-se a estrutura até que fique em uma posição fácil de distinguir as moléculas, com a M1 centralizada, e então uma imagem é capturada e as moléculas são identificadas, conforme foi exemplificado na Figura 2.6.

Continuando, são anotadas as informações da área de contato de cada molécula com a M1. Para isso, o poliedro é gerado novamente, mas apenas o trecho correspondente a cada molécula, uma por vez, o que permite exibir a área de contato entre a M1 e a M2, entre a M1 e a M3, e assim por diante.

Finalmente, os últimos passos do POP 01 [27] utilizam o programa Mercury [9]. A molécula da unidade assimétrica é expandida (com a função "Molecular Shell") até um tamanho que exiba todo o *cluster* supramolecular, e a estrutura é rotacionada para ter a mesma orientação preparada anteriormente no ToposPro [10]. Cada molécula é selecionada e o operador de simetria é anotado na planilha, identificando assim as moléculas (essa informação não está disponível no ToposPro). A estrutura completa é salva com a extensão .cif e depois cada molécula é salva no formato .xyz individualmente, da seguinte forma: removem-se todas as moléculas menos a M2; salva-se com o nome M2.xyz; abre-se novamente o arquivo .cif; repete-se o procedimento para a M3, M4, e assim por diante.

Quando esses passos são finalizados, significa que o POP 01 [27] foi integralmente cumprido e temos as informações das áreas de contato e os arquivos .xyz para cada molécula do *cluster*, que serão usados em um script que gera arquivos de entrada para os cálculos do Gaussian [26].

2.3 Poliedro de Voronoi-Dirichlet

O Poliedro de Voronoi-Dirichlet (VDP - *Voronoi-Dirichlet Polyhedron*) que é criado pelo ToposPro [10] nos passos do POP 01 [27] utiliza o conceito de diagramas de Voronoi para sua construção.

O diagrama de Voronoi (ou tesselação de Dirichlet), em gráficos 2D, é calculado identificando quais são as áreas que estão mais próximas de um ponto do que de qualquer outro ponto considerado [28]. Isso tem aplicabilidade em diversas áreas do conhecimento, por exemplo: na aviação, identificar rapidamente qual o aeroporto mais próximo de uma região no mapa [29]. Um exemplo de diagrama 2D de Voronoi pode ser visto na Figura 2.9 (imagem gerada com pontos aleatórios, a partir de um applet [30]).

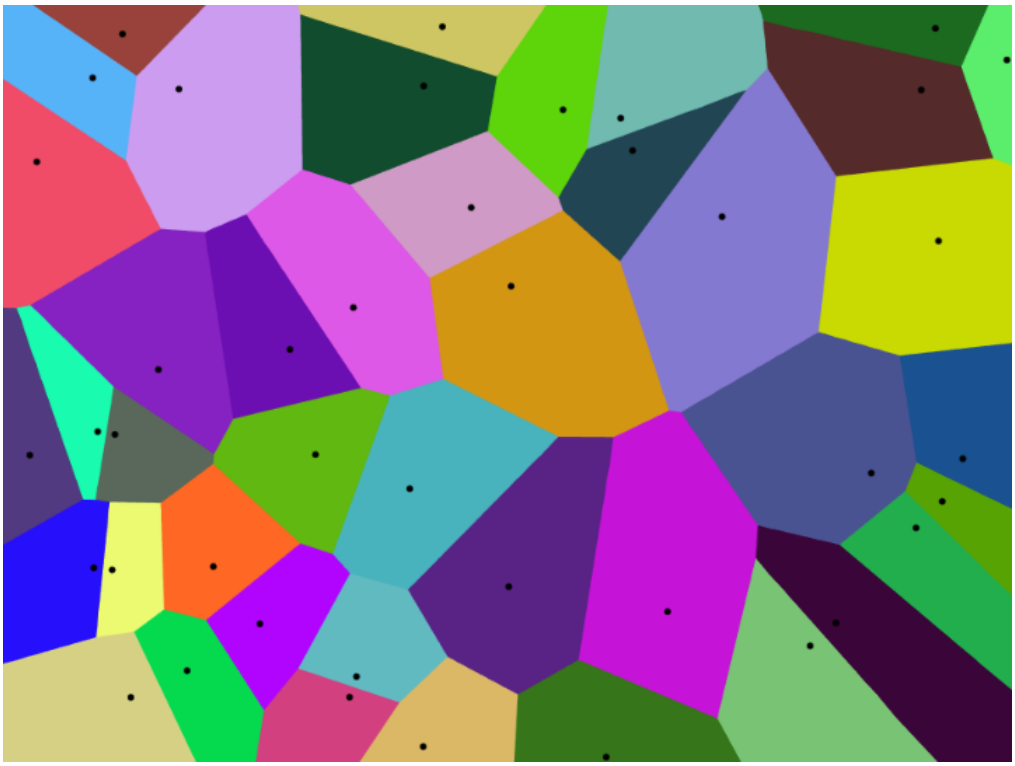

Figura 2.9: Exemplo de Diagrama 2D de Voronoi.

Esse conceito é expandido para o ambiente 3D (ou além), onde passa a delimitar o volume mais próximo de cada ponto. Um exemplo pode ser visto na Figura 2.10, onde um VDP foi gerado no ToposPro [10] selecionando-se apenas um átomo, gerando portanto um diagrama 3D de Voronoi em que o volume interno está mais próximo do átomo selecionado do que de qualquer outro átomo do cristal.

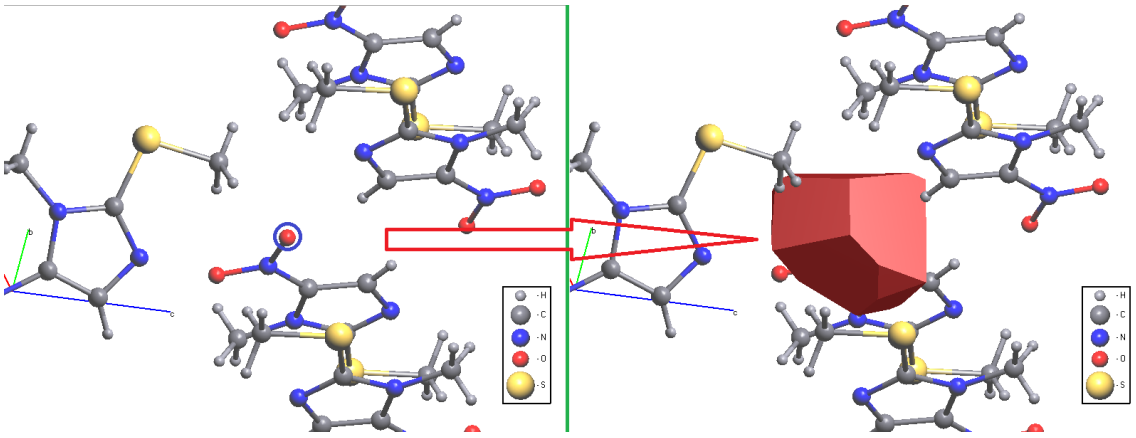

Figura 2.10: Diagrama 3D de Voronoi: Resultado da geração de um VDP, ao selecionar apenas um átomo.

Olhando esse mesmo diagrama por outro ângulo, conforme Figura 2.11, vemos

que a regra se aplica até mesmo para átomos da mesma molécula, podendo ter faces que cruzam ligações intermoleculares.

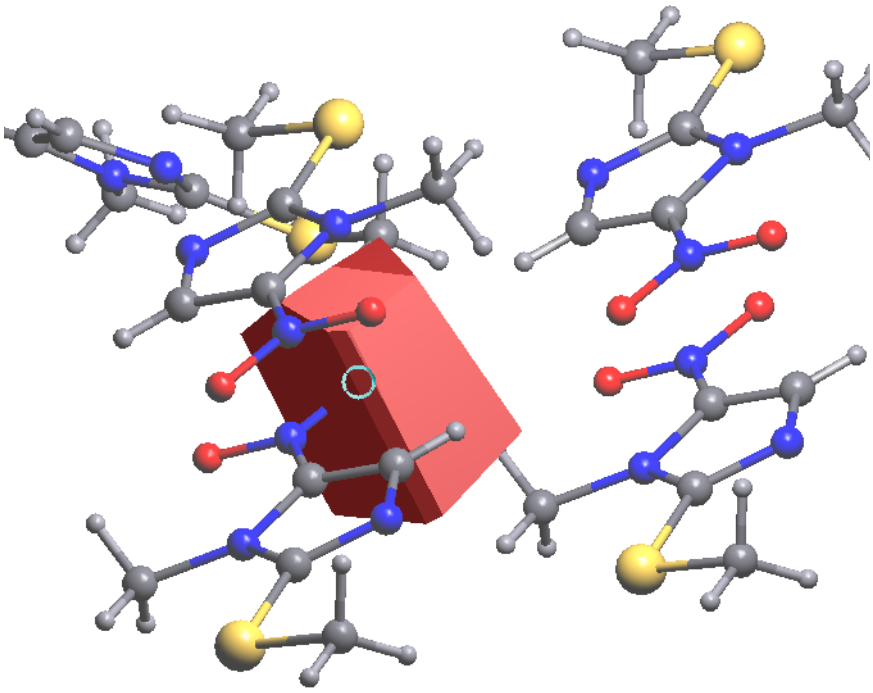

Figura 2.11: Diagrama da Figura 2.10 visto por outro ângulo, destacando a localização do átomo selecionado.

Isso não ocorre no VDP gerado pelo procedimento mencionado, porque a M1 inteira é selecionada, de forma que o poliedro resultante engloba todas as suas ligações intermoleculares.

Uma dúvida que surge naturalmente num primeiro momento é: se os diagramas de Voronoi consideram apenas pontos (sendo usada a posição do núcleo do átomo), que ponto seria utilizado para representar uma molécula inteira e gerar o VDP no seu entorno? Seria o ponto central da molécula? A resposta pode ser encontrada com uma observação cuidadosa do VDP gerado pelo ToposPro [10], a exemplo dos apresentados nas Figuras 2.7 e 2.8 na seção anterior, onde se pode verificar pelo seu formato que o poliedro não envolve um único ponto, mas sim todos os átomos da molécula, o que permite inferir que é gerado considerando átomo a átomo e utilizando somente as faces externas à molécula. Portanto, a geração do VDP consiste de gerar vários diagramas 3D de Voronoi (um para cada átomo da molécula) e juntar todos eles, desprezando as faces internas.

Outro passo do procedimento é a geração da parte do poliedro apenas entre a M1 e uma das moléculas. De fato, é possível gerar até mesmo uma única face do poliedro no ToposPro [10], por meio da seleção de apenas um átomo da M1 e um da outra molécula, conforme exemplificado na Figura 2.12.

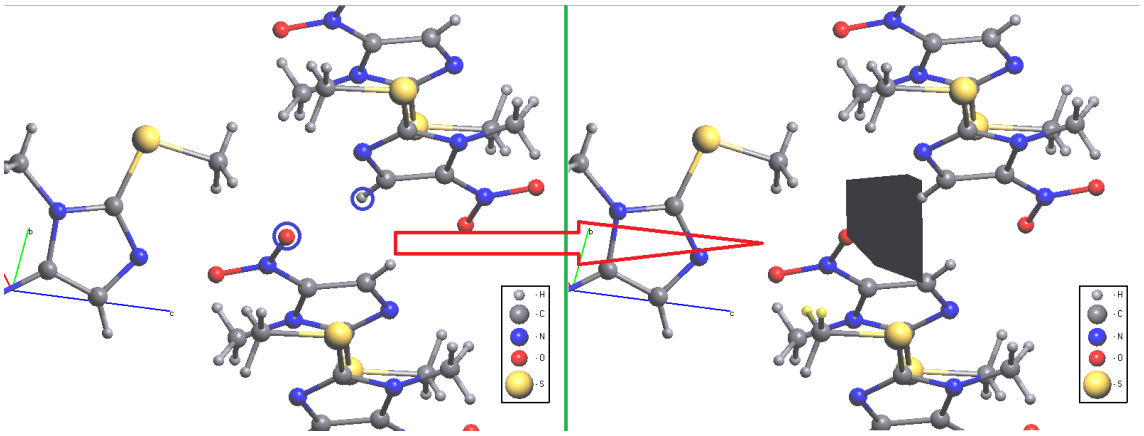

Figura 2.12: Geração de apenas uma face do VDP, situada entre os dois átomos destacados.

Essa face entre os dois átomos é a mesma quando é gerado o VDP completo, pois ela pode ser entendida como a região que está mais próxima desses dois átomos do que de qualquer outro, no plano intermediário entre eles. Para o objetivo deste trabalho, o dado que interessa é a área de cada face do poliedro, somando as áreas que se referem ao contato de átomos da M1 com átomos de uma mesma molécula e obtendo, assim, a área de contato da M1 com essa molécula.

Capítulo 3

Método Proposto

Para o método proposto, a linguagem Python com as bibliotecas CSD Python API [31] e Pyvoro [32] foram utilizadas. O IDE (*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado) utilizado foi o PyCharm [33].

O nome escolhido para o *software* foi PyCrystalSCD, que significa: *A Python Crystal's Supramolecular Cluster Demarcator*, pois a principal função dele é demarcar o *cluster* supramolecular do cristal.

Para executar os passos do POP 01 de forma automática, a ideia é que tudo o que é feito com os programas Mercury e ToposPro seja feito pelo próprio *software*. Porém, o que interessa é apenas a entrada (arquivo CIF com informações do cristal e escolha da M1) e as saídas (identificação das moléculas que compoem o *cluster* supramolecular, áreas de contato delas com a M1 e geração de arquivos .xyz para cada uma dessas moléculas). Assim, não é necessário reproduzir os passos em que o objetivo é apenas identificar visualmente as moléculas para distingui-las, nem procurar "buracos" no VDP. Desde que seja provida uma identificação inequívoca das moléculas e seja garantido que o poliedro estará completo, isso é suficiente.

Além disso, é importante frisar que a eficiência do *software*, embora desejável, não é fator determinante para seu sucesso. Se o tempo de execução não for excessivamente grande, o maior ganho está na possibilidade de deixar uma máquina calculando, em vez de exigir esforço e tempo humanos.

Assim, os passos necessários para o algoritmo do *software* são os seguintes:

- Ler o arquivo CIF;
- Listar as moléculas candidatas a serem a M1, permitindo que o usuário selecione qual delas será considerada, ou calcular automaticamente todas as opções;
- Expandir o cristal até garantir que contém todas as moléculas do *cluster*;
- Obter faces do VDP entre os átomos da M1 e os das outras moléculas, calculando suas áreas;

- Somar áreas referentes à mesma molécula e guardar a área de contato das MN moléculas com a M1;
- Exportar o resultado em uma planilha contendo: identificação das moléculas, operadores de simetria e áreas de contato com a M1;
- Salvar arquivos .xyz contendo os monômeros.

A forma como esses passos foram implementados será explicada em detalhes a seguir.

3.1 Desenvolvimento do *software*

Para realizar o desenvolvimento do *software* proposto com a linguagem Python, foram verificadas bibliotecas nessa linguagem que pudessem ler e interpretar arquivos CIF, e a mais conveniente encontrada foi uma provida pelo CSD [5], chamada de CSD Python API [31]. Essa biblioteca simplifica a leitura do arquivo CIF e o reconhecimento das moléculas.

Outra necessidade fundamental é a identificação das faces do poliedro de Voronoi-Dirichlet ao redor de toda a M1. Para isso, uma biblioteca Python chamada Pyvoro [32] deu uma enorme contribuição.

O código completo pode ser visto no Apêndice A.

3.1.1 Instalação das bibliotecas

Para utilizar essas bibliotecas no PyCharm [33], foi necessário instalá-las.

Para instalar a CSD Python API, um requisito era a instalação do lançamento CSDS (CSD-*System*) mais recente, conforme informado nas instruções de instalação [34]. Em seguida, foi suficiente utilizar, no PyCharm, o Python disponível na instalação (que, no caso do Windows na instalação padrão, é o arquivo C:\Program Files\CCDC\Python_API_2020\miniconda\python.exe).

Já a biblioteca Pyvoro, não funcionou na versão padrão, que teria a instalação mais fácil. Foi necessário:

- Baixar a versão presente no link:
  `https://github.com/joe-jordan/pyvoro/tree/feature/python3` [35]
- Copiar os arquivos libcrypto-1_1-x64.dll e libssl-1_1-x64.dll de:
  C:\Program Files\CCDC\Python_API_2020\miniconda\Library\bin
  para:
  C:\Program Files\CCDC\Python_API_2020\miniconda\DLLs
  (passo necessário para evitar um erro ao executar a instalação com pip install)

- No *prompt* de comando do Windows, como administrador, ir à pasta do pip do miniconda:
  cd C:\Program Files\CCDC\Python_API_2020\miniconda\Scripts
- Fazer a instalação referente à pasta onde o projeto foi baixado:
  pip install -e <pasta_pyvoro>
  (onde <pasta_pyvoro> deve ser substituído pelo caminho completo para a pasta onde o Pyvoro foi baixado)

Feito isso, as duas bibliotecas puderam ser utilizadas corretamente no PyCharm.

3.1.2 Leitura do arquivo CIF

Algumas das classes de objetos criados pelo CSD Python API são *Crystal* e *Molecule*. Quando lemos um arquivo CIF, um objeto da classe *Crystal* é criado e podemos acessar suas propriedades, como os seus parâmetros de rede e os operadores de simetria disponíveis. Já a classe *Molecule*, tem um conceito levemente diferente do usual: ela não representa apenas uma molécula (ligada por ligações covalentes), mas um conjunto de átomos, independente de estarem ligados ou não. No entanto, ela contém um atributo chamado *components*, que automaticamente verifica quais desses átomos estão ligados a outros por ligações covalentes e retorna uma lista com todas as moléculas formadas por esses átomos. Cada *component* também pertence à classe *Molecule*.

Para obter um objeto da classe *Molecule* que contém todos os átomos da unidade assimétrica, que correspondem ao operador de simetria (x,y,z), basta acessar um dos atributos da classe *Crystal*, conforme trecho de código abaixo:

```
unidade_assimetrica = crystal.asymmetric_unit_molecule
```

As moléculas candidatas a serem a M1 são aquelas que contém esses átomos, mas, para obtê-las, nem sempre é suficiente obter os *components* da unidade assimétrica. Existem casos em que a molécula completa também contém átomos gerados por outros operadores de simetria, que podem estar até mesmo na cela unitária vizinha. Assim, é necessário antes expandir o cristal.

3.1.3 Expansão do cristal

A expansão do cristal, além de completar as moléculas candidatas a serem a M1, é necessária para obter as demais moléculas do *cluster*. Ela pode ser realizada por meio de uma função muito útil provida por essa biblioteca: *crystal.packing()*, que retorna uma *Molecule* contendo os átomos do cristal dentro de um limite determinado (parâmetro *box_dimensions*). E ela permite passar o parâmetro *inclusion='AnyAtomIncluded'*, que faz com que, se uma molécula estiver incompleta

(apenas parte da molécula estiver dentro do limite estabelecido), a molécula inteira será retornada. Assim, obtemos facilmente todas as moléculas que podem fazer parte do *cluster* supramolecular, desde que escolhamos um limite suficiente para garantir que todas elas estarão incluídas.

O parâmetro *box_dimensions* permite informar multiplicadores para os eixos da cela. O padrão é *box_dimensions* = ((0, 0, 0), (1, 1, 1)), que limita ao espaço da cela unitária. Considerando o conhecimento adquirido sobre como a cela unitária gera o restante do cristal, a estratégia escolhida para ter essa garantia seguiu este raciocínio: as moléculas da unidade assimétrica sempre ocupam a cela unitária ou, no máximo, parte de suas celas vizinhas, e nunca vão além do limite destas. Logo, estarão contidas com certeza na expansão obtida ao aplicar *box_dimensions* = ((-1, -1, -1), (2, 2, 2)), e essa região também irá conter as moléculas geradas pela aplicação direta dos operadores de simetria iniciais (ou uma de suas translações).

Se expandirmos o cristal ainda mais, fazendo *box_dimensions* = ((-2, -2, -2), (3, 3, 3)), é mais que suficiente para garantir que todas as moléculas do entorno da M1 estarão contidas, pois garantimos que até mesmo a própria M1 foi repetida em todas as direções e, se fosse expandido ainda mais, qualquer molécula além das já encontradas estaria por trás de outra desse grupo ou distante demais, não fazendo parte da primeira esfera de coordenação. Para garantir ainda com mais certeza, também foi utilizado o parâmetro *inclusion='AnyAtomIncluded'*. Assim, o comando executado ficou sendo:

```
cristal_expandido = crystal.packing(box_dimensions=((-2, -2, -2), (3,
    3, 3)), inclusion='AnyAtomIncluded')
```

A Figura 3.1 mostra um exemplo para esclarecer a ideia. A M1 está parcialmente fora dos limites da cela unitária, mas se repete duas vezes por translação para todos os lados e, ainda que em um dos eixos ela estivesse completamente fora da cela, ainda se repetiria ao menos uma vez. As celas além desses limites conterão apenas novas repetições, que estarão seguramente fora do entorno necessário para obter o *cluster* supramolecular, garantindo assim que não ocorrerão buracos no poliedro de Voronoi-Dirichlet.

3.1.4 Cálculo dos operadores de simetria

Uma das funcionalidades solicitadas para o *software* foi a geração de uma planilha que resumisse as informações calculadas. Uma das informações importantes para ajudar a identificar rapidamente cada molécula é o operador de simetria que gerou seus átomos.

Para obter a lista de operadores definidos, o CSD Python API provê o atributo *crystal.symmetry_operators*, que retorna os operadores como texto. Esses textos

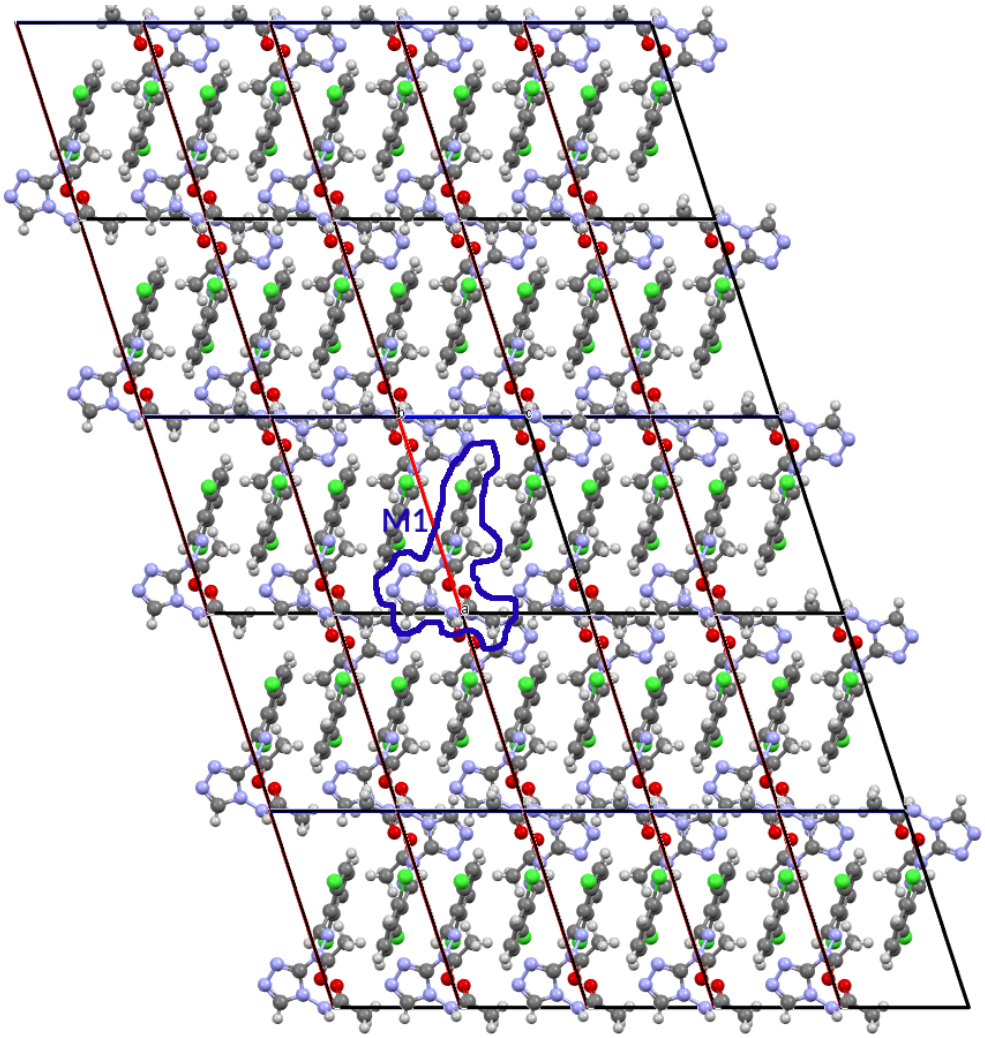

Figura 3.1: Exemplo de aplicação do método *crystal.packing(box_dimensions=((-2, -2, -2), (3, 3, 3)), inclusion='AnyAtomIncluded')*. Editado a partir de imagem obtida no Mercury para o cristal com código CCDC AABHTZ, posicionado de frente para o eixo y, destacando a M1.

podem ser convertidos em uma matriz de rotação e um vetor de translação, a fim de facilitar a aplicação do operador à posição de um átomo (basta multiplicar a matriz de rotação pelo vetor de posição e somar com o vetor de translação). Para isso, utilizam-se os métodos *symmetry_rotation* e *symmetry_translation*, providos pela biblioteca. O código que obtém essas informações ficou desta forma:

```
operadores = []
for symmop in crystal.symmetry_operators:
    matriz_rotacao = np.reshape(crystal.symmetry_rotation(symmop), (3,
        3))
    operadores.append({'rotacao': matriz_rotacao,
                       'translacao': crystal.symmetry_translation(
                           symmop)})
```

Porém, essa é apenas a lista de operadores originais. Há infinitas opções de operadores, pois cada coordenada fracionária pode ser somada a qualquer número inteiro, resultando numa translação para outra cela unitária.

Para descobrir qual o operador que gerou determinado átomo a partir de sua posição na unidade assimétrica, primeiro foi necessário utilizar um artifício que fizesse uma normalização da posição, ou seja: a posição correspondente, de forma que todas as coordenadas fracionárias sejam números entre 0 e 1. Assim, basta ver se a posição normalizada do átomo corresponde à da aplicação de um operador ao átomo correspondente da unidade assimétrica. O operador em que isso se verificar é o que gerou o átomo, sendo necessário corrigir apenas o vetor de translação em seguida.

O código da normalização tem a seguinte estrutura:

```
def posicao_atomo_normalizada(posicao_atomo):
    # Obtém a posição do átomo equivalente na cela unitária original -
        coordenadas fracionárias no intervalo [0, 1)
    posicao_corrigida = posicao_atomo.copy()
    for i in range(0, 3):
        parte_inteira = math.trunc(posicao_atomo[i])
        posicao_corrigida[i] -= parte_inteira
        if posicao_atomo[i] < 0:
            posicao_corrigida[i] += 1
    return posicao_corrigida
```

E o código que identifica o operador que gerou um átomo, a partir dos operadores originais e do átomo original (o átomo com mesmo rótulo, na unidade assimétrica), foi definido como:

```
def encontra_operador_do_atomo(operadores_originais, atom,
    atomo_original):
    posicao_atomo_atual = np.array(atom.fractional_coordinates)
    for operador in operadores_originais:
        # Verifica se alguma translação deste operador resulta na posiç
            ão do átomo
        posicao_apos_operador_inicial = aplica_operador(atomo_original,
            operador)
        if mesmo_ponto(posicao_atomo_normalizada(
            posicao_apos_operador_inicial),
                       posicao_atomo_normalizada(posicao_atomo_atual)):
            # Encontrado o operador original, corrige o vetor de
                translação
            diferenca_posicao = posicao_atomo_atual -
                posicao_apos_operador_inicial
            return {'rotacao': operador['rotacao'], 'translacao':
                operador['translacao'] + diferenca_posicao}
```

3.1.5 Remoção de átomos inválidos

Em alguns casos, um operador de simetria pode gerar átomos na mesma posição. Isso é resultado da estratégia de refinamento realizada pelo autor da estrutura. No entanto, para nossa proposta, é necessário retirar os átomos adicionais para poder realizar o método. A Figura 3.2, obtida a partir do Mercury [9], ilustra esse tipo de caso na estrutura com código CCDC QQQCIV04. Em destaque, estão os átomos que se "chocam".

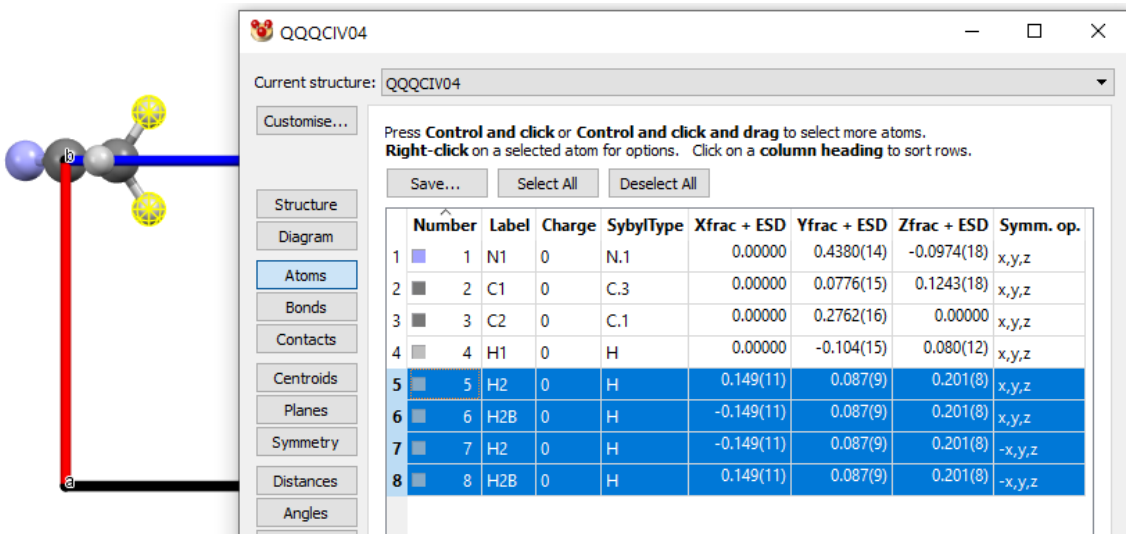

Figura 3.2: Cristal que gera átomos inválidos. Código CCDC: QQQCIV04

A unidade assimétrica desse cristal tem 6 átomos. Para 4 deles, os operadores (x,y,z) e (-x,y,z) geram os mesmos tipos de átomos nas mesmas posições, pois x = 0 (Figura 3.2). Isso não é um problema: tanto o Mercury quanto o CSD Python API desconsideram o átomo gerado pelo segundo operador, apresentando-o apenas uma vez naquela posição. Mas os outros dois são hidrogênios com rótulos H2 e H2B, e um é gerado na posição do outro.

Assim, por terem rótulos diferentes, eles ainda são incluídos, mesmo havendo choque de posições. Isso provoca problemas nos cálculos posteriores do *software*, além de não ter sentido químico, fazendo-se necessário remover esses átomos repetidos de todas as moléculas antes de prosseguir.

Para identificar quais as posições inválidas, considerando como válida sempre a do primeiro operador, percorre-se os átomos da unidade assimétrica em cada operador e verifica-se se algum deles choca a posição normalizada com a de um operador anterior, com um rótulo diferente. O código que obtém essas posições foi definido como:

```
def obtem_posicoes_normalizadas_invalidas(unidade_assimetrica,
    operadores):
```

```
    # Guarda posições normalizadas inválidas (repetição átomos de
        labels diferentes na mesma posição)
    posicoes_normalizadas_invalidas = defaultdict(list)
    posicoes_normalizadas_validas = defaultdict(list)

    for operador in operadores:
        # Guarda posições normalizadas válidas e dos átomos em desordem
        for atom in unidade_assimetrica.atoms:
            posicao_normalizada = posicao_atomo_normalizada(
                aplica_operador(atom, operador))
            if outro_atomo_possui_ponto(atom.label, posicao_normalizada
                , posicoes_normalizadas_validas):
                posicoes_normalizadas_invalidas[atom.label].append(
                    posicao_normalizada)
            else:
                posicoes_normalizadas_validas[atom.label].append(
                    posicao_normalizada)
    return posicoes_normalizadas_invalidas


def outro_atomo_possui_ponto(label_atomo_atual,
    posicao_normalizada_atual, posicoes_normalizadas_validas):
    for label in posicoes_normalizadas_validas.keys():
        # Quando o label é o mesmo, a geração do mesmo ponto não é um
            problema
        if label != label_atomo_atual:
            if possui_ponto(posicao_normalizada_atual,
                posicoes_normalizadas_validas[label]):
                return True
    return False
```

3.1.6 Preparação do poliedro de Voronoi-Dirichlet

Após identificar a M1, crescer o cristal e remover os átomos inválidos, a principal funcionalidade necessária para viabilizar o *software* é a identificação das faces do poliedro de Voronoi-Dirichlet ao redor da M1.

A ideia básica do algoritmo é identificar o plano intermediário entre um ponto e cada um dos seus vizinhos, cortando os planos encontrados e considerando apenas o mais próximo, até que não haja mais cortes. Apesar de clara a ideia, colocar isso em linhas de código seria bem trabalhoso e sujeito a erros que poderiam não ser percebidos facilmente.

Para isso, a biblioteca Pyvoro [32], que utiliza a biblioteca Voro++ [36], foi utilizada. Ela calcula todos os vértices de um diagrama 3D de Voronoi-Dirichlet, dados os pontos e uma delimitação do espaço considerado. Um exemplo pode ser

visto na Figura 3.3, obtida no *site* do Voro++ [37].

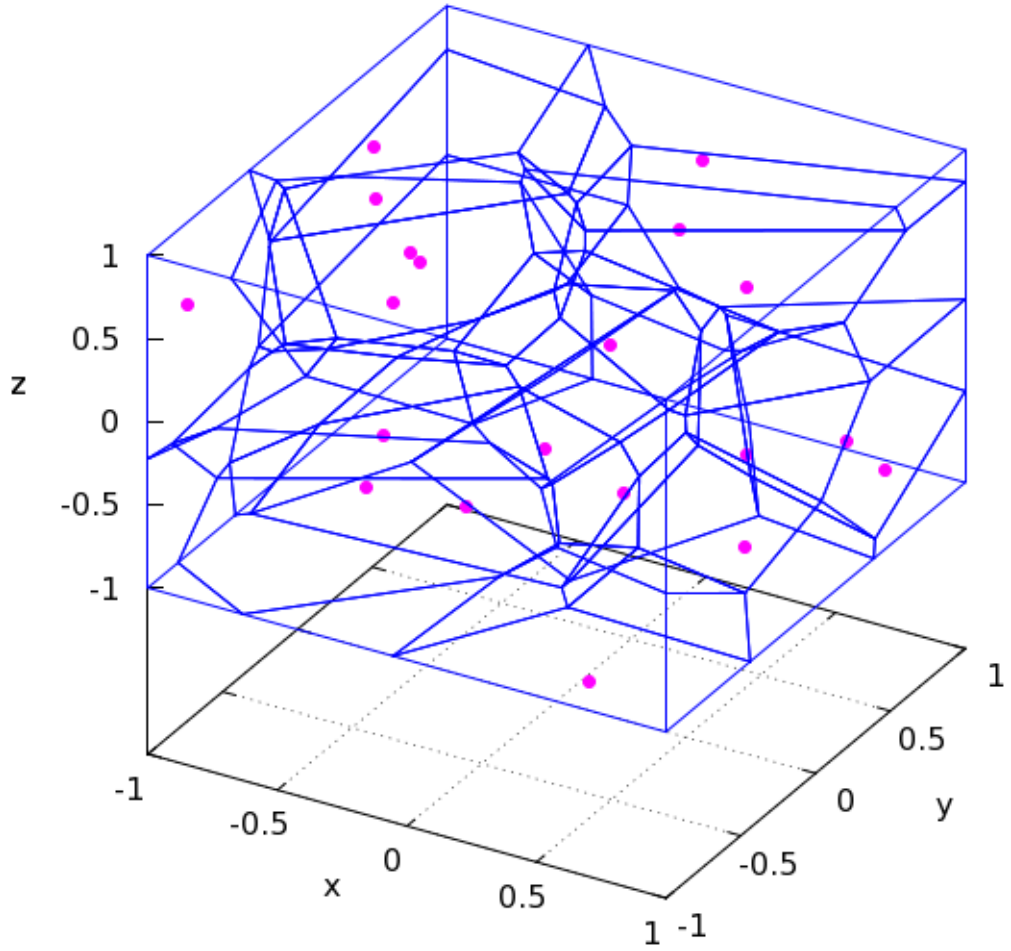

Figura 3.3: Exemplo de diagrama 3D de Voronoi calculado com a biblioteca Voro++.

Apesar de a Voro++ ter sido desenvolvida em C++, ela pode ser acessada por linha de comando e foi utilizada pela Pyvoro para permitir que um código Python acesse suas funcionalidades principais. O método *compute_voronoi* recebe como entrada um *array* de posições dos pontos, os limites mínimo e máximo para as coordenadas x, y e z e a distância máxima a considerar entre dois pontos adjacentes. Como retorno, ele informa todas as faces do diagrama, com seus vértices e a informação de quais são os pontos envolvidos (pois cada face do diagrama 3D situa-se no plano intermediário entre um par de pontos adjacentes).

Assim, foram passadas todas as posições dos átomos das moléculas encontradas no passo anterior, estabelecido um limite que mantém todos esses pontos e, para a distância máxima, foi escolhida a maior das medidas dos parâmetros de rede **a**, **b** e **c**, acrescido de 0,1 Å. Isso é suficiente para encontrar o mesmo átomo após uma translação de uma unidade fracionária em qualquer das direções, sendo também suficiente para encontrar todos os possíveis contatos do átomo.

```
distancia_maxima_contato = max(crystal.cell_lengths) + 0.1
```

Apesar de isso significar que muito mais moléculas do que as necessárias farão parte do cálculo, aumentando o tempo de processamento das informações, essa foi a melhor maneira encontrada de garantir que todo o *cluster* supramolecular está abrangido. Como explicado anteriormente, a eficiência não é a maior prioridade neste momento. E não seria possível apenas verificar se há buracos no poliedro, como realizado no procedimento usando o ToposPro [10], pois esses buracos só existem porque o programa considera algum átomo que não foi selecionado na geração do poliedro. Se tal átomo não estivesse sendo considerado, o poliedro estaria fechado naquela região, pois os planos das faces vizinhas se estenderiam até se encontrar (consequentemente, tais áreas estariam maiores do que deveriam). Logo, somente a garantia de incluir todos os átomos do *cluster* permite o cálculo correto do poliedro.

Após executar o método *compute_voronoi*, o algoritmo percorre os átomos da M1 de um em um, fazendo um mapeamento dos átomos que possuem área de contato com eles e quais são as moléculas a que pertencem. Para calcular as áreas, foram utilizadas as informações retornadas pelo Pyvoro para cada átomo (identificado como um ponto), que são: as coordenadas dos vértices de cada face do poliedro que envolve aquele átomo; e qual é o outro átomo envolvido naquela face.

Por exemplo, retomando parte da Figura 2.12, que mostra a face de um poliedro formada entre um átomo de oxigênio de uma molécula e um de hidrogênio de outra, a Figura 3.4 mostra os vértices V_1 a V_6 dessa face. O que o Pyvoro retorna em relação a esse átomo de oxigênio, por exemplo, são as coordenadas desses 6 vértices e do outro átomo envolvido (que está equidistante a essa face - no caso, esse átomo de hidrogênio). E o mesmo para todas as demais faces ao redor desse átomo (sendo desconsideradas as faces entre dois átomos da própria M1, já que não fazem parte do poliedro).

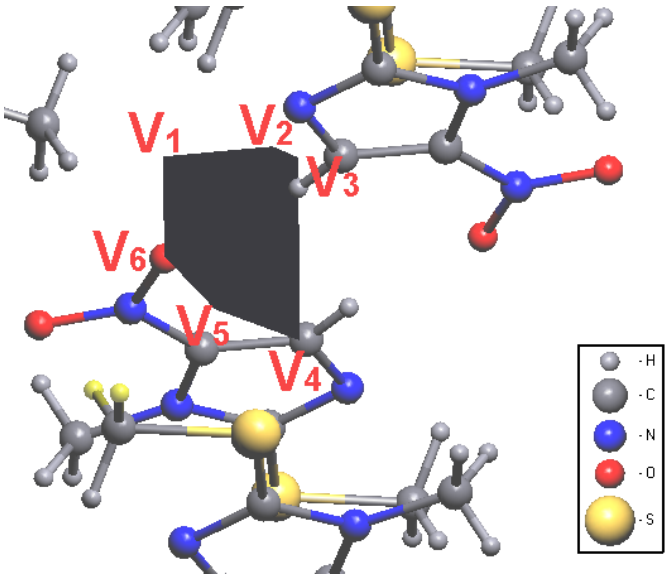

Figura 3.4: Vértices de uma face do VDP.

A partir das coordenadas dos vértices, é possível calcular a área de contato entre os dois átomos. E as áreas correspondentes a átomos de uma mesma molécula em contato com a M1 são somadas. Assim, após terminado o procedimento para todos os átomos da M1, está calculada a área de todo o poliedro que circunda a M1 e separadas as áreas de contato com cada molécula, que é o resultado principal pretendido.

Para calcular a área a partir dos vértices de uma face do poliedro, foi utilizada uma abordagem deduzida a partir da decomposição de triângulos, que permite a utilização de uma fórmula concisa para esse cálculo [38]. As figuras utilizadas na explicação dessa abordagem, dada a seguir, foram obtidas e adaptadas a partir da mesma fonte [38].

Inicialmente, consideremos um polígono 2D simples com N vértices, onde cada aresta liga dois vértices consecutivos (V_i e V_{i+1}). Para calcular sua área, consideremos os vetores que vão de um ponto P qualquer (interno ou externo ao polígono) aos vértices, conforme esquematizado na Figura 3.5.

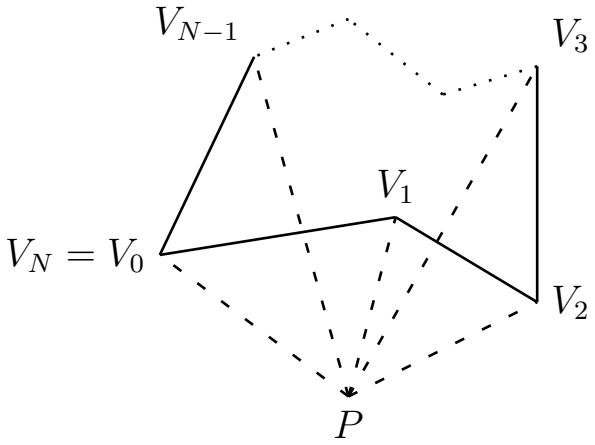

Figura 3.5: Polígono 2D genérico, cuja área será calculada considerando os vetores que vão do ponto P aos vértices.

A área de cada triângulo PV_iV_{i+1} pode ser calculada por meio do produto vetorial $(V_i - P) \times (V_{i+1} - P)$, pois o vetor resultante tem uma magnitude igual ao dobro do valor buscado. Embora a área seja um valor em módulo, iremos considerar o sinal das áreas intermediárias calculadas, sendo positivas quando o movimento de V_i para V_{i+1} for no sentido anti-horário e negativas quando horário (conforme o sentido do vetor que a calcula). Representaremos áreas negativas em vermelho e positivas em azul.

Assim, iremos somar as áreas de todos os triângulos e o resultado final será a área do polígono. Começando pelo triângulo PV_0V_1, que percorre um sentido horário, temos uma área negativa.

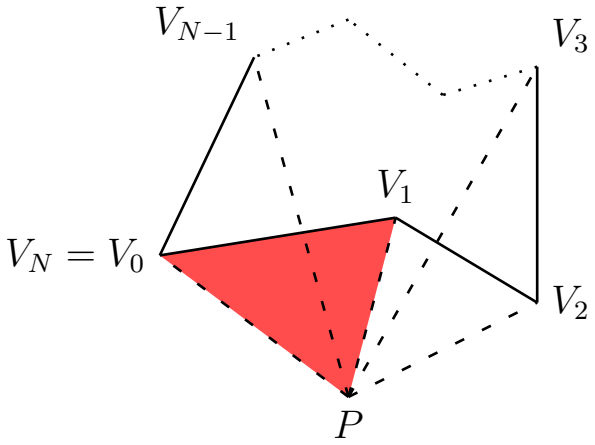

Figura 3.6: Incluindo a área do triângulo PV_0V_1.

Novamente, o triângulo PV_1V_2 tem área negativa, externa ao polígono.

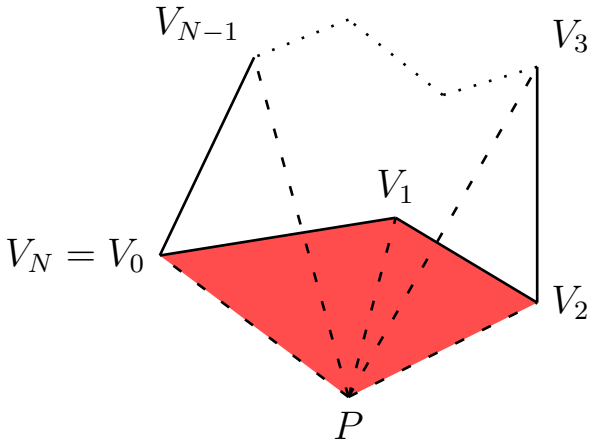

Figura 3.7: Somando a área do triângulo PV_1V_2.

Em seguida, o triângulo PV_2V_3 tem área positiva, que ao ser somada cancela parte da área negativa anterior.

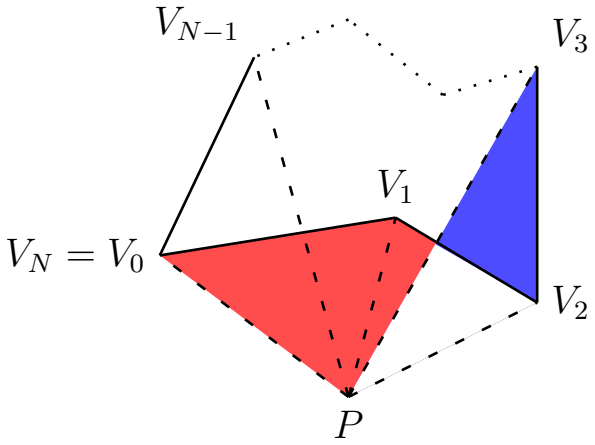

Figura 3.8: Somando a área do triângulo PV_2V_3.

Esse padrão se mantém nos triângulos seguintes, com área positiva que acrescenta à área interna ao polígono e anula da área externa.

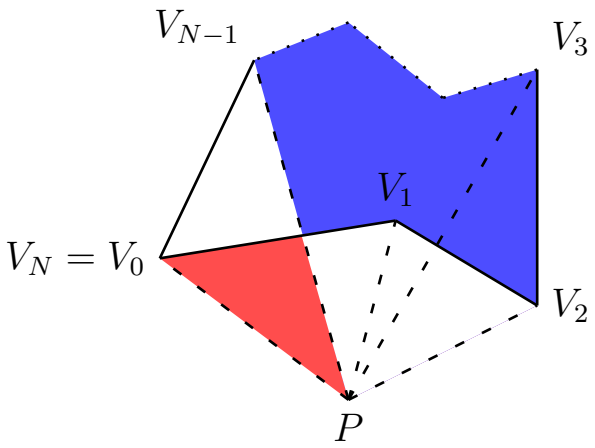

Figura 3.9: Somando áreas até o vértice V_{N-1}.

Finalmente, com o triângulo P$V_{N-1}V_N$, resta apenas a área do polígono, com valor positivo. O valor seria negativo se os vértices estivessem dispostos no sentido horário. Portanto, deve ser aplicado módulo ao resultado final.

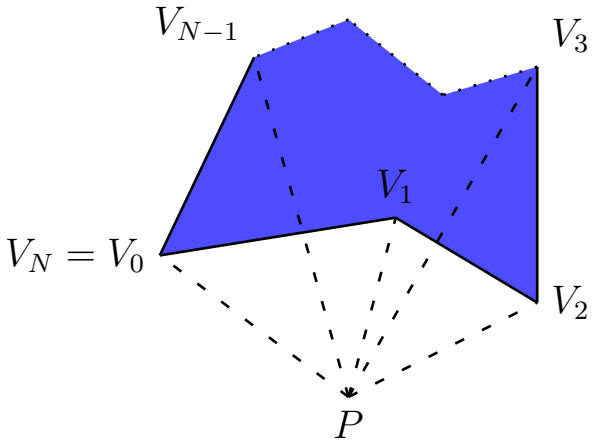

Figura 3.10: Finalizando a soma da área do polígono.

A mesma lógica é aplicada no caso de um polígono plano $\mathbf{\Omega}$ no $\mathbb{R}^3$, conforme representado na Figura 3.11. Também são utilizados triângulos que vão dos vértices a um ponto P qualquer, mas esses triângulos serão projetados para o plano $\mathscr{P}$ que contém o polígono. Para esse cálculo, usaremos também o vetor $\mathbf{n}$, que é unitário e normal a $\mathscr{P}$.

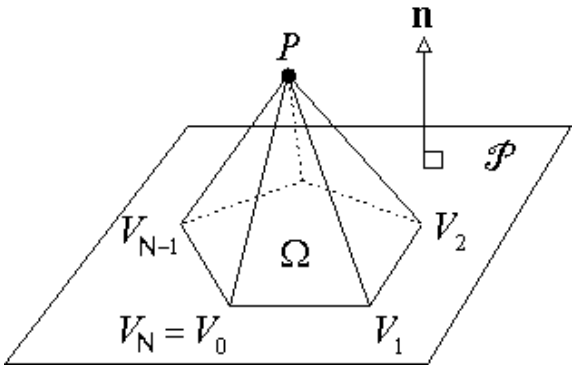

Figura 3.11: Esquematização de um polígono 3D e recursos auxiliares para calcular sua área.

A área de cada triângulo PV_iV_{i+1} corresponde à magnitude de um vetor $\boldsymbol{\alpha_i}$, que é assim calculado:

$$\boldsymbol{\alpha_i} = \frac{(V_i - P) \times (V_{i+1} - P)}{2}$$

Na Figura 3.12, vemos que o ângulo θ entre $\boldsymbol{\alpha_i}$ e $\mathbf{n}$ é o mesmo que ocorre entre o triângulo PV_iV_{i+1} e o plano $\mathscr{P}$. Assim, a área projetada em $\mathscr{P}$ corresponde à projeção de $\boldsymbol{\alpha_i}$ em $\mathbf{n}$. Como $\mathbf{n}$ é unitário, esse valor é obtido por meio do produto escalar $\mathbf{n} \cdot \boldsymbol{\alpha_i}$.

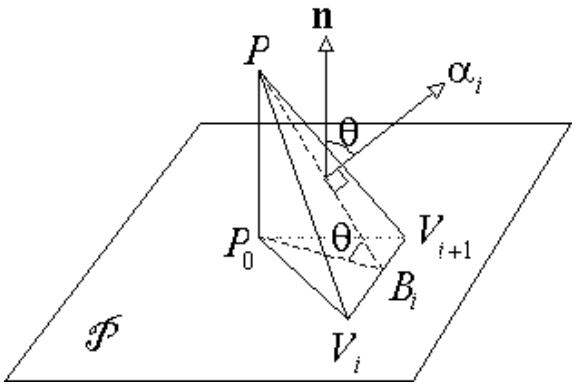

Figura 3.12: Demonstração dos ângulos envolvidos no cálculo da área projetada.

Esse produto resulta numa área com sinal: positivo quando o movimento de V_i para V_{i+1} for no sentido anti-horário (a partir do ponto de vista apontado pela normal), negativo quando horário. Portanto, para a área de um polígono completo $\mathbf{\Omega}$ com N vértices conforme esquematizado na Figura 3.13, a mesma ideia utilizada no polígono 2D pode ser seguida, somando-se áreas positivas e negativas e resultando na área do polígono.

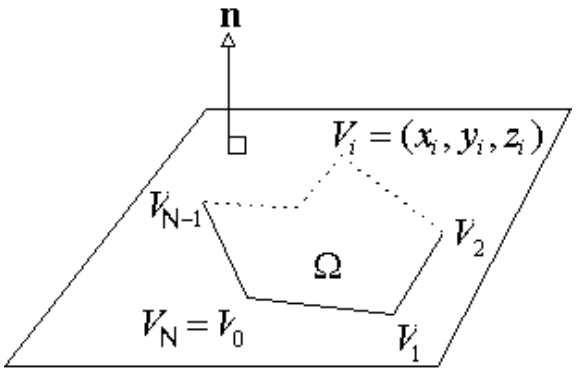

Figura 3.13: Exemplo de polígono 3D e a normal ao plano que o contém.

Assim, a área do polígono $\mathbf{\Omega}$ pode ser calculada desta forma

$$\mathrm{A}(\mathbf{\Omega}) = \left| \sum_{i=0}^{N-1} \mathbf{n} \cdot \boldsymbol{\alpha_i} \right| = \left| \frac{\mathbf{n}}{2} \cdot \sum_{i=0}^{N-1} [(V_i - P) \times (V_{i+1} - P)] \right|.$$

Como o ponto P pode ser escolhido arbitrariamente, escolhemos $P = (0, 0, 0)$, de forma que $V_i - P = V_i$ (vetor que vai da origem ao vértice V_i). Dessa forma, obtemos esta fórmula concisa

$$\mathrm{A}(\mathbf{\Omega}) = \frac{1}{2} \left| \mathbf{n} \cdot \sum_{i=0}^{N-1} (V_i \times V_{i+1}) \right|.$$

Para aplicar essa fórmula, um pré-requisito é que os vértices do polígono estejam

ordenados de forma que V_i e V_{i+1} sempre estejam ligados por uma aresta do polígono. Mas os vértices de cada face dos diagramas de Voronoi retornados pelo Pyvoro já vêm ordenados dessa forma, pois eles são obtidos por meio do método `face_vertices` do Voro++, que, como é um código aberto, está disponível para consulta [39]. Assim, pode ser verificado que, antes de obter o próximo vértice, é utilizado o método `cycle_up`, que, conforme documentação, serve para obter a próxima aresta a partir do vértice, no sentido anti-horário. Logo, o próximo vértice pertence a essa aresta, garantindo que o pré-requisito seja atendido.

Além dos vértices, também é necessário o vetor **n**. Mas ele pode ser obtido facilmente utilizando os 3 primeiros vértices e fazendo o produto vetorial $(V_1 - V_0) \times (V_2 - V_0)$, que resulta num vetor normal ao plano que, por sua vez, é dividido pela própria magnitude para se tornar unitário, conforme mostrado na Figura 3.14.

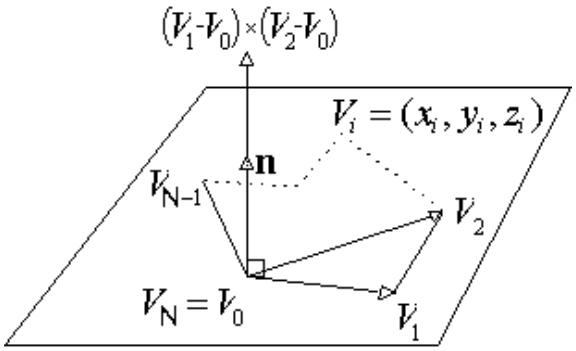

Figura 3.14: Obtenção do vetor normal unitário.

As operações com vetores são facilitadas em Python com o uso da biblioteca numpy, cuja instalação é trivial e é de praxe ser resumida como *np*. Assim, o código que calcula a área a partir dos vértices (já ordenados) ficou relativamente simples, sendo quase uma aplicação direta das fórmulas apresentadas. Para obter facilmente o vértice $V_N = V_0$, foi considerado como índice o resto da divisão por N (operador % do Python). Esse código é apresentado abaixo.

```
def calcula_area(vertices):
    num_vertices = len(vertices)
    if num_vertices < 3:  # não forma um plano - não tem area
        return 0

    # Obtém a normal ao plano: produto vetorial entre os vetores (v1 -
        v0) e (v2 - v0)
    normal = np.cross(np.subtract(vertices[1], vertices[0]), np.
        subtract(vertices[2], vertices[0]))
    # Divide pela magnitude (norma) para obter o vetor unitário
    normal_unitaria = normal / np.linalg.norm(normal)
```

```
    # Soma (com sinal) dos produtos vetoriais V(i) x V(i+1), onde V(i)
        é o vetor que vai da origem até o vértice V(i)
    total = [0, 0, 0]
    for i in range(num_vertices):
        vi1 = vertices[i]
        vi2 = vertices[(i+1) % num_vertices]
        prod = np.cross(vi1, vi2)
        total += prod

    # O módulo do produto escalar do somatório com a normal unitária ao
        plano resulta no dobro da área do polígono
    result = np.dot(total, normal_unitaria)
    return abs(result/2)
```

3.1.7 Preparação dos arquivos de saída

As saídas necessárias para este *software* são uma planilha que resume os resultados e os arquivos .xyz com as posições dos átomos de cada molécula (monômeros). Porém, também foram considerados desejáveis a geração de arquivos .mol2 com esses monômeros e com os dímeros formados pela M1 junto com cada uma das demais moléculas do *cluster*, para facilitar a visualização, caso necessário.

Para a planilha, o formato de arquivo gerado é .csv (*Comma Separated Value*) [40]. Esse tipo de arquivo é simples de ser gerado, sendo ainda mais facilitado pelo módulo *csv*, que está disponível junto com a instalação do Python. Assim, a geração da planilha utiliza um simples *array* bidimensional, conforme trecho de código abaixo:

```
        conteudo_csv = [["Symmetry Code", "Dimer", "Area"]]

        area_total = sum([area_molecula['area'] for area_molecula in
            area_contato_por_molecula])
        operadores_molecula = encontra_operadores(molecula_m1,
            operadores, mapa_unidade_assimetrica)
        conteudo_csv.append([operadores_molecula, nome_m1, round(
            area_total, 4)])
...
        with open(nome_arquivo_completo + ".csv", 'w', newline='') as
            arquivo_csv:
             writer = csv.writer(arquivo_csv, delimiter=';')
             writer.writerows(conteudo_csv)
```

Para os arquivos .xyz, eles seguem uma lógica simples e puderam ser gerados "do zero" [41]. Para os .mol2, foi utilizada uma funcionalidade provida pelo CSD Python API: *io.MoleculeWriter*. Uma pequena correção foi necessária após gerar o arquivo,

porque os objetos *Molecule* não possuem informações do cristal, que são importantes neste caso. A correção consistiu de obter a linha com essa informação diretamente a partir do objeto *Crystal*, e então acrescentar aos arquivos .mol2 gerados ao final. O código pode ser visualizado abaixo:

```
# Linha que antecede a linha com as informações do cristal, em arquivos
    .mol2
MARCADOR_INFO_CRISTAL_MOL2 = "@<TRIPOS>CRYSIN"

def obtem_informacoes_do_cristal_para_mol2(crystal):
    linhas_mol2 = crystal.to_string('mol2').splitlines()
    linha_info_cristal = None
    achou_linha = False
    for linha in linhas_mol2:
        if achou_linha:
            linha_info_cristal = linha
        if linha.startswith(MARCADOR_INFO_CRISTAL_MOL2):
            achou_linha = True
    return linha_info_cristal

def salva_molecula_em_arquivos_mol2_e_xyz(molecula, diretorio,
    nome_arquivo, linha_info_cristal):
    nome_arquivo_completo = os.path.join(diretorio, nome_arquivo)
    # Cria o arquivo mol2
    with io.MoleculeWriter(nome_arquivo_completo + ".mol2") as
        mol_writer:
        mol_writer.write(molecula)
    # Acrescenta as informações do cristal
    with open(nome_arquivo_completo + ".mol2", 'a') as arquivo_mol2:
        arquivo_mol2.write(MARCADOR_INFO_CRISTAL_MOL2 + "\n")
        arquivo_mol2.write(linha_info_cristal + "\n")
    # Cria o arquivo xyz
    with open(nome_arquivo_completo + ".xyz", 'w') as arquivo_xyz:
        # Número de átomos e descrição
        arquivo_xyz.write("%d\n%s\n" % (len(molecula.atoms),
            nome_arquivo))
        # Átomos e suas posições
        for atom in molecula.atoms:
            arquivo_xyz.write("{:4} {:17.12f} {:17.12f} {:17.12f}\n".
                format(
                atom.atomic_symbol, atom.coordinates.x, atom.
                    coordinates.y, atom.coordinates.z))

...
linha_info_cristal = obtem_informacoes_do_cristal_para_mol2(crystal)
```

Capítulo 4

Resultados e Discussões

Uma vez calculada a área de contato entre as moléculas, foi feita a comparação entre os resultados obtidos com o PyCrystalSCD e os obtidos com cálculos manuais. Primeiramente, foi verificado o caso do cristal utilizado no POP 01 [27], (código CCDC: LERKIE). Tal documento apresenta, ao final, uma planilha com os resultados da análise manual desse cristal. A Tabela 4.1 reproduz o conteúdo dessa planilha e apresenta, ao lado, as áreas calculadas pelo PyCrystalSCD, para comparação.

LERKIE			
Simmetry Code	Dimer	Å^2	Áreas calculadas:
x,y,z	M1	268,49	268.4896
-1+x,y,z	M1...M2	46,36	46.3574
1+x,y,z	M1...M3	46,36	46.3574
2-x,-y,-z	M1...M4	2,04	2.039
1-x,1-y,-z	M1...M5	28,15	28.1463
2-x,1-y,-z	M1...M6	36,98	36.9773
1-x,2-y,-z	M1...M7	1,29	1.2938
-1+x,1+y,z	M1...M8	5,07	5.0661
x,1+y,z	M1...M9	7,77	7.7676
1/2-x,1/2+y,1/2-z	M1...M10	22,06	22.0568
1.5-x,1/2+y,1/2-z	M1...M11	18,77	18.7686
1/2-x,-1/2+y,1/2-z	M1...M12	22,06	22.0568
1.5-x,-1/2+y,1/2-z	M1...M13	18,77	18.7686
x,-1+y,z	M1...M14	7,77	7.7676
1+x,-1+y,z	M1...M15	5,07	5.0661
N: 14	**Total:**	**268,52**	N: 14

Tabela 4.1: Planilha com as áreas de contato obtidas no POP 01 para a estrutura com código CSD LERKIE, confrontada com as áreas calculadas pelo PyCrystalSCD para a mesma estrutura.

Observa-se que, para esse caso, os resultados do PyCrystalSCD foram certeiros. Os valores arredondados encontrados para as áreas de contato foram exatamente os mesmos, assim como o valor de N (Número de coordenação molecular, ou seja, o número de moléculas com alguma área de contato com a M1). E a soma das áreas, ao considerar as casas decimais adicionais, resultou em 268,4896 Å^2, que corresponde à área total do poliedro anotada na primeira linha de dados da planilha. Isso explica a discrepância entre esse valor e o da última linha da planilha como sendo provocada pela soma de valores já arredondados.

Para testar mais casos e checar a corretude, em um primeiro momento foram utilizados outros 52 arquivos CIF, de estruturas que já têm artigos publicados pelo NUQUIMHE, todas com Z'=1. O PyCrystalSCD foi executado para todos eles e as diferenças foram avaliadas com o apoio de alunos. Os resultados observados foram os seguintes:

- Em quase todos os casos, o algoritmo calculou as áreas corretamente, com exceção de duas das estruturas, obtendo um nível de acerto de 96,15%;
- Em dois casos, o algoritmo calculou um valor de N maior, sendo verificado que havia uma das faces com área muito pequena (0,0019 Å^2 e 0,0008 Å^2), arredondada para 0,00 Å^2. Os alunos aplicaram novamente o procedimento para esses casos e encontraram esses pequenos "buracos" no poliedro (vide Figura 4.1);

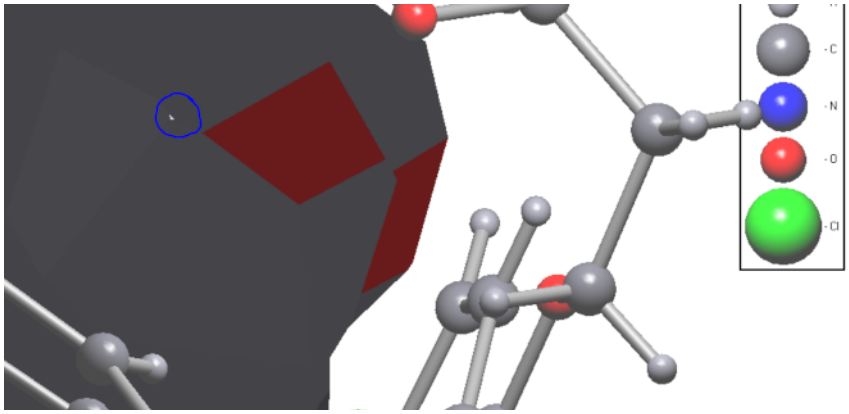

Figura 4.1: "Buraco" no poliedro de Voronoi-Dirichlet com área muito pequena (0,000783 Å^2). Código CCDC: AHUQEC.

- As duas estruturas com erro tiveram a causa principal identificada: ambas apresentavam desordem, que é a geração de um mesmo átomo em mais de uma posição possível na mesma molécula. Em uma delas, a desordem era devido à existência de dois átomos ocupando a mesma coordenada. Na outra, hidrogênios apareciam duplicados.

Quanto ao tempo de execução apenas da verificação das áreas de contato, foi feita uma comparação com o tempo despendido, no processo manual, por alunos com experiência variada (Tabela 4.2). 7 alunos realizaram novos cálculos em 9 estruturas com $Z' = 1$, dos quais dois foram desconsiderados por se afastarem muito do padrão: um deles por ter feito poucos sistemas e outro por ter tempo médio de apenas 5 minutos, devido à marcação do tempo em estágio diferente. A amostra com os 5 alunos restantes fornece uma boa visão média dessa atividade.

Apesar de o PyCrystalSCD ser executado apenas em uma máquina e depender dos programas que estão executando de maneira paralela, o tempo medido também fornece uma aproximação média. O *software* foi, em média, 80 vezes mais rápido para realizar a mesma atividade (Tabela 4.2).

Aluno [a]	Estrutura									Média
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
	Tempo (min:seg)									
1	15:34	11:51	23:27	17:15	09:41	11:50	11:49	18:29	11:49	14:38
2	21:02	11:18	20:40	13:07	11:36	14:40	15:29	18:07	18:09	16:00
3	14:09	12:09	21:49	16:46	11:08	12:02	17:34	20:42	15:45	15:47
4	17:00	17:00	14:00	12:00	16:00	15:00	23:00	10:00	21:00	16:06
5	13:37	13:48	23:57	16:45	12:16	12:54	19:35	27:01	16:33	17:22
Média	16:16	13:13	20:46	15:10	12:08	13:17	17:29	18:51	16:39	15:59
Software	00:08	00:15	00:11	00:16	00:18	00:15	00:12	00:05	00:07	00:12
Vezes Mais Rápido	122	50	113	58	41	52	84	216	149	80

[a] Experiência média do aluno: 1 (4 anos), 2 (2 anos) e 3-5 (1 ano).

Tabela 4.2: Comparação entre os tempos de execução manual e do *software*.

Dentre as estruturas testadas, 4 apresentavam simetria, de forma que apenas metade da molécula aparece na cela unitária e a outra metade é gerada por um operador de simetria. Um exemplo pode ser visto na Figura 4.2 (imagem obtida no Mercury [9]). Como a expansão do cristal obtém moléculas inteiras, esses casos também foram calculados corretamente.

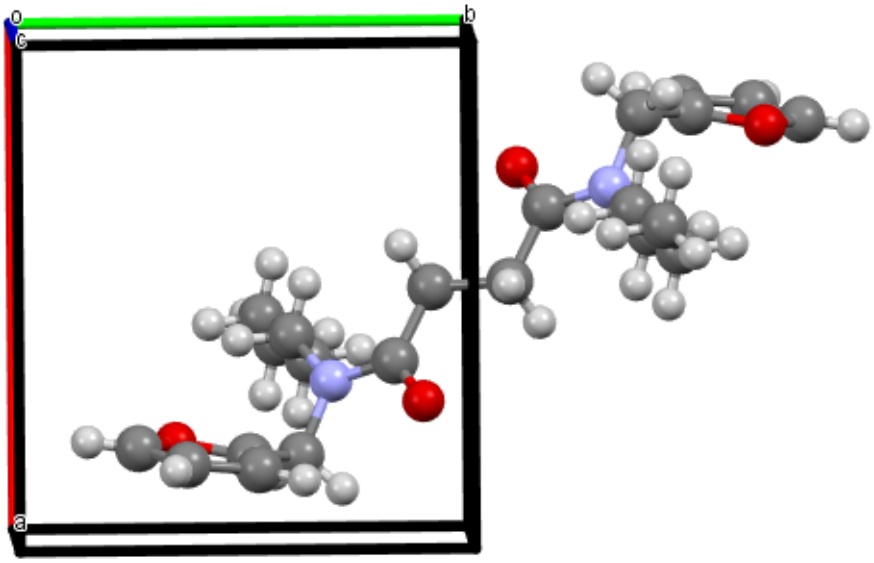

Figura 4.2: Exemplo de molécula que apresenta simetria de uma cela para outra. Código CCDC: CIDQOY.

A análise do primeiro *dataset* com 52 moléculas possibilitou confirmar a corretude do PyCrystalSCD e mapear as dificuldades a serem corrigidas para os casos pontuais de desordem e de estruturas com Z'>1. Após uma tentativa de correção dos casos de desordem, os dois casos problemáticos foram corrigidos, mas alguns dos casos que funcionavam passaram a ter resultado divergente. Então essa solução teve que ser adaptada.

A atualização do código resolveu o problema do átomo com posição inválida (mencionado no tópico 3.1.5) e também passou a considerar casos com Z'>1, ao mesmo tempo que manteve funcionando as demais 50 estruturas que já eram calculadas corretamente. No entanto, não houve tempo hábil para buscar uma solução para o outro caso com desordem sem prejudicar as demais estruturas. Mais dois casos, que apresentam Z'>1, lograram êxito nessa solução.

O código final, além de incluir essas correções, passou a realizar a exportação dos arquivos de saída, conforme informado no tópico 3.1.7. Exemplos dos arquivos gerados podem ser observados no Apêndice B.

Após a conclusão da nova versão, o *software* foi confrontado novamente com o procedimento manual. Nessa comparação final, diferente da anterior (Tabela 4.2), foi considerado todo o processo de construção do *cluster* supramolecular, ou seja: identificação da M1, construção do *cluster*, retirada das áreas de contato, exportação dos dados dos dímeros em planilha com os respectivos rótulos, áreas de contato e a rastreabilidade dos códigos de simetria. Além disso, a exportação dos arquivos em formato .xyz e .mol2 dos monômeros e dímeros, para ser possível a utilização nos cálculos de química quântica. Com essas etapas consideradas, é possível dimensionar

o resultado final do PyCrystalSCD.

O *dataset* foi definido em 6 estruturas com moléculas de distintos tamanhos e diferentes níveis de complexidade para construção dos *clusters*. Das seis estruturas, cinco apresentam Z'=1 (uma delas com simetria) e uma apresenta Z'=2. Como resultado, temos a construção de 7 *clusters*. O *dataset* foi realizado manualmente por discentes do NUQUIMHE-UFSM com diferentes níveis de experiência e confrontado com o *software*. Os resultados podem ser observados na Tabela 4.3.

Aluno [a]	Estrutura						Total
	1	**2**	3 [b]	**4**	**5**	6 [c]	
	Tempo (hora:min:seg)						
1	00:21:04	00:15:54	00:24:07	00:17:11	00:23:16	00:40:30	02:22:02
2	00:30:03	00:22:00	00:16:14	00:25:27	00:32:40	00:56:08	03:02:32
3	00:17:29	00:20:19	00:18:39	00:17:02	00:31:25	00:48:56	02:33:50
4	00:16:20	00:24:02	00:35:00	00:27:00	00:23:45	00:39:47	02:45:54
5	00:25:00	00:47:00	00:43:00	00:34:00	00:49:00	01:39:00	04:57:00
6	00:42:00	00:50:00	00:46:00	01:01:00	00:50:00	02:00:00	06:09:00
7	00:29:00	00:42:00	00:29:00	00:23:00	00:35:00	01:05:00	03:43:00
Média	00:25:51	00:31:36	00:30:17	00:29:14	00:35:01	01:07:03	03:39:03
Software	00:00:07	00:00:58	00:00:23	00:00:10	00:00:14	00:00:37	00:02:29

[a] Experiência média do aluno: 1-2 (4 anos), 3-4 (2 anos), 5-6 (1 ano) e 7 (<3 meses).

[b] Estrutura simétrica.

[c] Estrutura com Z'=2.

Tabela 4.3: Comparação entre os tempos de execução manual e do *software* para todo o processo de construção do *cluster* supramolecular.

A partir dos resultados da comparação do processo completo (Tabela 4.3) é possível observar que o PyCrystalSCD foi 88 vezes mais rápido que a execução manual realizada pelos discentes, como demonstra a Figura 4.3. Com as mesmas 3 horas e 39 minutos do tempo médio total no processo manual, seria possível realizar pelo *software* o cálculo de 617 *clusters* supramoleculares (Figura 4.3).

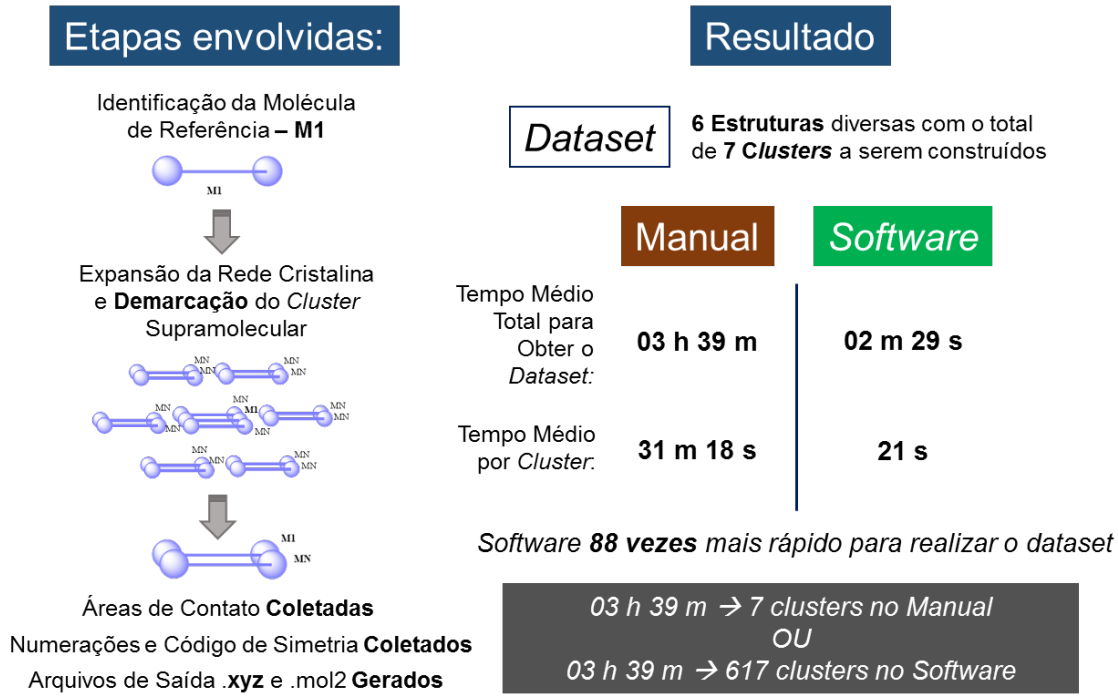

Figura 4.3: Comparação entre a execução manual e do *software* para todo o processo de construção do *cluster* supramolecular.

4.1 Melhorias para o futuro

O PyCrystalSCD está atendendo de maneira eficaz o propósito de contribuir significativamente na velocidade da análise dos cristais e geração dos arquivos de entrada para cálculos de energia. Porém, algumas melhorias são desejáveis:

- Melhorar a solução aplicada para casos com $Z'>1$. Nesta versão, foi considerada cada molécula que contém átomos da unidade assimétrica como sendo uma possível M1. Falta incluir os casos em que a M1 é formada por uma combinação dessas moléculas (se $Z'=2$, por exemplo, com moléculas A e B, está sendo feito o cálculo com a M1 sendo M1A e M1B, mas não M1AB);

- Implementar soluções para casos de arquivos CIF com desordem. Uma ideia é passar a verificar se há mais ligações do que as permitidas para determinado átomo, identificando então se a exclusão dos átomos gerados por um dos operadores resolve o problema;

- Melhorar o tempo de processamento por meio da diminuição do distanciamento limite considerado na geração de diagramas de Voronoi;

- Conectar à máquina virtual que tem o Gaussian instalado e calcular automaticamente as energias, preenchendo o resultado na planilha;
- Definir regras que identifiquem automaticamente se uma sequência de cristais analisados está seguindo determinado padrão, permitindo analisar toda a base do CSD, e além!

Capítulo 5

Conclusões

O *software* desenvolvido cumpriu, de forma eficaz, o objetivo de obter a demarcação do *cluster* supramolecular, a partir da leitura de arquivos CIF, de cristais orgânicos. Além disso, foi possível alcançar um procedimento em que são obtidas as áreas de contato de cada dímero que compõe o *cluster* supramolecular, junto à rastreabilidade dos seus respectivos códigos de simetria e a exportação dos arquivos em formato .xyz para posteriores cálculos de química quântica.

O cálculo das áreas de contato apresentou resultados que coincidiram com os obtidos no protocolo manual, atingindo 98,3% de acerto, sendo que o único caso testado que não alcançou o resultado esperado apresentava desordem na estrutura.

Sendo assim, este trabalho busca contribuir na construção do conhecimento na área de engenharia de cristais ao permitir estudos em larga escala de forma rápida e automatizada.

Referências Bibliográficas

[1] BRAGA, D. "Crystal engineering, Where from? Where to?" *Chemical Communications*, pp. 2751–2754, 2003. doi: 10.1039/B306269B.

[2] LEHN, J.-M. "Supramolecular Chemistry - Scope and Perpectives". 1987. Disponível em: <https://www.nobelprize.org/prizes/chemistry/1987/lehn/lecture/>. Acesso em: 15/12/2020.

[3] DE ARAUJO, G. L. B., PITALUGA, A., ANTONIO, S. G., et al. "Polimorfismo na produção de medicamentos", *Revista de Ciências Farmacêuticas Básica e Aplicada*, v. 33, pp. 27–36, 2012.

[4] BRUM, J. A. "Determinação experimental das estruturas cristalinas e espaço recíproco". 2014. Disponível em: <https://sites.ifi.unicamp.br/brum/files/2014/01/F888_JAB_1s2010_P2_cap3.pdf>.

[5] GROOM, C. R., BRUNO, I. J., LIGHTFOOT, M. P., et al. "The Cambridge Structural Database", *Acta Cryst.*, v. B72, pp. 171–179, 2016. doi: 10.1107/S2052520616003954.

[6] CHAUMONT, C., MOBIAN, P., KYRITSAKASB, N., et al. "Synthesis, topology and energy analysis of crystalline resorcinol-based oligophenylene molecules with various symmetries", *CrystEngComm*, v. 15, pp. 6845–6862, 2013.

[7] DAY, G. M., COOPER, T. G., CRUZ-CABEZA, A. J., et al. "Significant progress in predicting the crystal structures of small organic molecules - a report on the fourth blind test", *CrystEngComm*, v. 65, pp. 107–125, 2009. doi: 10.1107/S0108768109004066.

[8] MARTINS, M. A. P., FRIZZO, C. P., MARTINS, A. C. L., et al. "Energetic and topological approach for characterization of supramolecular clusters in organic crystals", *RSC Adv*, v. 4, pp. 44337–44349, 2014. doi: 10.1039/C4RA06040G.

[9] MACRAE, C. F., SOVAGO, I., COTTRELL, S. J., et al. "Mercury 4.0: from visualization to analysis, design and prediction", *J. Appl. Cryst.*, v. 53, 2020.

[10] BLATOV, V. A., SHEVCHENKO, A. P., PROSERPIO, D. M. "Applied topological analysis of crystal structures with the program package ToposPro", *Cryst. Growth Des.*, v. 14, pp. 3576–3586, 2014.

[11] MARTINS, M. A. P., MEYER, A. R., TIER, A. Z., et al. "Proposal for crystallization of 3-amino-4-halo-5-methylisoxazoles: an energetic and topological approach", *CrystEngComm*, v. 17, pp. 7381–7391, 2015.

[12] ZIMMER, G. C., PAGLIARI, A. B., BENDER, C. R., et al. "Insights on conformation in the solid state: a case study – s-cis and/or s-trans crystallization of 5(3)-aryl-3(5)-carboxyethyl-1-tert-butylpyrazoles", *CrystEngComm*, v. 20, pp. 5154–5168, 2018.

[13] COPETTI, J. P. P., SALBEGO, P. R. S., ORLANDO, T., et al. "Substituent effects on the crystallization mechanisms of 7-chloro-4-substituted-quinolines", *CrystEngComm*, 2020. doi: 10.1039/D0CE00214C.

[14] MARTINS, M. A. P., PAGLIARI, A. B., BELLADONA, A. L., et al. "Supramolecular self-assembly and thermodynamic properties of 5-aryl-1-(1,1-dimethylethyl)-1H-pyrazoles in the crystalline state", *J. Mol. Struct.*, v. 1195, pp. 570–581, 2019.

[15] MARTINS, M. A. P., SALBEGO, P. R. S., DE MORAES, G. A., et al. "Understanding the crystalline formation of triazene N-oxides and the role of halogen$\cdots\pi$ interactions", *CrystEngComm*, v. 20, pp. 96–112, 2018.

[16] PAGLIARI, A. B., ORLANDO, T., SALBEGO, P. R. S., et al. "Supramolecular Packing of a Series of N-Phenylamides and the Role of NH$\cdots$O=C Interactions", *ACS Omega*, v. 3, pp. 13850–13861, 2018.

[17] ORLANDO, T., SALBEGO, P. R. S., FARIAS, F. F. S., et al. "Crystallization Mechanisms Applied to Understand the Crystal Formation of Rotaxanes", *European J. Org. Chem.*, v. 2019, pp. 3451–3463, 2019.

[18] FRIZZO, C. P., BENDER, C. R., TIER, A. Z., et al. "Energetic and topological insights into the supramolecular structure of dicationic ionic liquids", *CrystEngComm*, v. 17, pp. 2996–3004, 2015.

[19] MARTINS, M. A. P., HÖRNER, M., BECK, J., et al. "Polymorphism in an 18-membered macrocycle: an energetic and topological approach to understand the supramolecular structure", *CrystEngComm*, v. 18, pp. 3866–3876, 2016.

[20] ORLANDO, T., SALBEGO, P. R. S., TASCHETTO, C. L. R., et al. "Polymorphism in a Rotaxane Molecule: Intra- and Intermolecular Understanding", *Cryst. Growth Des.*, v. 19, pp. 1021–1030, 2019.

[21] SALBEGO, P. R. S., BENDER, C. R., HÖRNER, M., et al. "Insights on the Similarity of Supramolecular Structures in Organic Crystals Using Quantitative Indexes", *ACS Omega*, v. 3, pp. 2569–2578, 2018.

[22] SALBEGO, P. R. S., BENDER, C. R., ORLANDO, T., et al. "Supramolecular Similarity in Polymorphs: Use of Similarity Indices (I^X)", *ACS Omega*, v. 4, pp. 9697–9709, 2019.

[23] SMART, L. E., MOORE, E. A. *Solid State Chemistry: An Introduction*. 4 ed. Boca Raton, CRC Press, 2012.

[24] "CCDC - CSD Entry: WERPOY". 1994. Disponível em: <https://www.ccdc.cam.ac.uk/structures/Search?Ccdcid=WERPOY&DatabaseToSearch=Published>.

[25] BARBOSA, K. F. *Determinação por Difração de Raios X da Estrutura Molecular do 1L-1,2,3,4,5-Ciclohexanopentol*. Dissertação de M.Sc., Universidade Estadual de Goiás, Anápolis, GO, Brasil, 2009. Disponível em: <http://livros01.livrosgratis.com.br/cp138912.pdf>.

[26] FRISCH, M. J., TRUCKS, G. W., SCHLEGEL, H. B., et al. "Gaussian˜16 Revision C.01". 2016. Gaussian Inc. Wallingford CT.

[27] MARTINS, M. A. P., COPETTI, J. P. P. "Procedimento Operacional Padrão - 01 - Obtenção de cluster supramolecular". 2019.

[28] MUMM, M. "Voronoi Diagrams", *The Mathematics Enthusiast*, v. 1, pp. 44–55, 2004. Disponível em: <https://scholarworks.umt.edu/tme/vol1/iss2/4/>.

[29] GORCZYNSKI, G. "Proximity Analysis with Voronoi Diagrams - Mapping US International Airports". 2017. Disponível em: <https://tableaupicasso.com/proximity-analysis-voronoi-diagrams/>.

[30] BEUTEL, A. "Interactive Voronoi Diagram Generator with WebGL". 2020. Disponível em: <http://alexbeutel.com/webgl/voronoi.html>.

[31] "CSD Python API". 2020. Disponível em: <https://www.ccdc.cam.ac.uk/solutions/csd-system/Components/csd-python-api/>.

[32] "Pyvoro - 2D and 3D Voronoi tessellations: a python entry point for the voro++ library". 2015. Disponível em: <https://pypi.org/project/pyvoro/>.

[33] "PyCharm: O IDE Python para desenvolvedores profissionais". 2020. Disponível em: <https://www.jetbrains.com/pt-br/pycharm/>.

[34] "Installation notes - CSD Python API documentation". 2020. Disponível em: <https://downloads.ccdc.cam.ac.uk/documentation/API/installation_notes.html>.

[35] "Pyvoro for Python3". 2015. Disponível em: <https://github.com/joe-jordan/pyvoro/tree/feature/python3>. Acesso em: 20/10/2020.

[36] "Voro++ - A 3D Voronoi cell software library". 2013. Disponível em: <http://math.lbl.gov/voro++/>.

[37] "random_points.cc – The Voronoi diagram for random points in a cube". 2013. Disponível em: <http://math.lbl.gov/voro++/examples/random_points/>.

[38] SUNDAY, D. "Area of Triangles and Polygons". 2012. Disponível em: <http://geomalgorithms.com/a01-_area.html>.

[39] "Voro++: cell.cc Source File". 2011. Disponível em: <http://math.lbl.gov/voro++/doc/refman/cell_8cc_source.html>.

[40] REPICI, D. J. "How To: The Comma Separated Value (CSV) File Format". 2010. Disponível em: <http://creativyst.com/Doc/Articles/CSV/CSV01.htm>.

[41] "XMol XYZ File Format". 2020. Disponível em: <https://www.ch.ic.ac.uk/ectoc/ectoc-3/instruct/xmolxyz.html>.

Apêndice A

Código

```
from ccdc import io
import pyvoro
import numpy as np
import math
import os.path
from collections import defaultdict
import tkinter as tk
from tkinter import filedialog
import csv
import time


# Linha que antecede a linha com as informações do cristal, em arquivos .mol2
MARCADOR_INFO_CRISTAL_MOL2 = "@<TRIPOS>CRYSIN"


def mesmo_ponto(ponto1, ponto2):
    return np.isclose(ponto1[0], ponto2[0]) and np.isclose(ponto1[1], ponto2[1])
        ↪ and np.isclose(ponto1[2], ponto2[2])


def possui_ponto(ponto_novo, array_de_pontos):
    for ponto_array in array_de_pontos:
        if mesmo_ponto(ponto_novo, ponto_array):
            return True
    return False


def distancia_pontos(ponto1, ponto2):
    return np.sqrt((ponto1[0]-ponto2[0])**2+(ponto1[1]-ponto2[1])**2+(ponto1[2]-
        ↪ ponto2[2])**2)


def calcula_area(vertices):
    n = len(vertices)
    if n < 3:  # não forma um plano - não tem area
        return 0

    # Obtém a normal ao plano: produto vetorial entre os vetores (v1 - v0) e (v2 -
        ↪ v0)
    normal = np.cross(np.subtract(vertices[1], vertices[0]), np.subtract(vertices
        ↪ [2], vertices[0]))
```

```
    # Divide pela magnitude (norma) para obter o vetor unitário
    normal_unitaria = normal / np.linalg.norm(normal)

    # Soma (com sinal) dos produtos vetoriais V(i) x V(i+1), onde V(i) é o vetor
        ↪ que vai da origem até o vértice V(i)
    total = [0, 0, 0]
    for i in range(n):
        vi1 = vertices[i]
        vi2 = vertices[(i+1) % n]
        prod = np.cross(vi1, vi2)
        total += prod

    # O módulo do produto escalar do somatório com a normal unitária ao plano
        ↪ resulta no dobro da área do polígono
    result = np.dot(total, normal_unitaria)
    return abs(result/2)


def corrige_maior_menor_coordenadas(coordenadas_novo_atomo, menor_xyz, maior_xyz):
    for indice, coord_novo in enumerate(coordenadas_novo_atomo):
        if coord_novo < menor_xyz[indice]:
            menor_xyz[indice] = coord_novo
        if coord_novo > maior_xyz[indice]:
            maior_xyz[indice] = coord_novo


def inclui_posicoes_atomos_da_molecula(molecula, posicoes_atomos,
    ↪ molecula_por_posicao, maior_xyz, menor_xyz):
    for atom in molecula.atoms:
        c = atom.coordinates
        posicoes_atomos.append([c.x, c.y, c.z])
        corrige_maior_menor_coordenadas(c, menor_xyz, maior_xyz)
        molecula_por_posicao.append(molecula)


def obtem_posicoes_normalizadas_invalidas(unidade_assimetrica, operadores):
    # Guarda posições normalizadas inválidas (repetição átomos de labels diferentes
        ↪  na mesma posição)
    posicoes_normalizadas_invalidas = defaultdict(list)
    posicoes_normalizadas_validas = defaultdict(list)

    for operador in operadores:
        # Guarda posições normalizadas válidas e dos átomos em desordem
        for atom in unidade_assimetrica.atoms:
            posicao_normalizada = posicao_atomo_normalizada(aplica_operador(atom,
                ↪ operador))
            if outro_atomo_possui_ponto(atom.label, posicao_normalizada,
                ↪ posicoes_normalizadas_validas):
                posicoes_normalizadas_invalidas[atom.label].append(
                    ↪ posicao_normalizada)
            else:
                posicoes_normalizadas_validas[atom.label].append(
                    ↪ posicao_normalizada)
    return posicoes_normalizadas_invalidas


def outro_atomo_possui_ponto(label_atomo_atual, posicao_normalizada_atual,
    ↪ posicoes_normalizadas_validas):
    for label in posicoes_normalizadas_validas.keys():
```

```
        # Quando o label é o mesmo, a geração do mesmo ponto não é um problema
        if label != label_atomo_atual:
            if possui_ponto(posicao_normalizada_atual,
                ↪ posicoes_normalizadas_validas[label]):
                return True
    return False


def posicao_atomo_normalizada(posicao_atomo):
    # Obtém a posição do átomo equivalente na cela unitária original - coordenadas
        ↪ fracionárias no intervalo [0, 1)
    posicao_corrigida = posicao_atomo.copy()
    for i in range(0, 3):
        parte_inteira = math.trunc(posicao_atomo[i])
        posicao_corrigida[i] -= parte_inteira
        if posicao_atomo[i] < 0:
            posicao_corrigida[i] += 1
    return posicao_corrigida


def aplica_operador(atom, operador):
    # Aplica operador de simetria (matriz de rotação e vetor de translação)
    posicao_atomo = np.dot(operador['rotacao'], atom.fractional_coordinates) +
        ↪ operador['translacao']
    return posicao_atomo


def remove_atomos_invalidos(molecula, posicoes_normalizadas_invalidas):
    atomos_removidos = []
    for atom in molecula.atoms:
        posicao_corrigida = posicao_atomo_normalizada(np.array(atom.
            ↪ fractional_coordinates))
        if possui_ponto(posicao_corrigida, posicoes_normalizadas_invalidas[atom.
            ↪ label]):
            molecula.remove_atom(atom)
            atomos_removidos.append(atom)
    return atomos_removidos


def obtem_informacoes_do_cristal_para_mol2(crystal):
    linhas_mol2 = crystal.to_string('mol2').splitlines()
    linha_info_cristal = None
    achou_linha = False
    for linha in linhas_mol2:
        if achou_linha:
            linha_info_cristal = linha
        if linha.startswith(MARCADOR_INFO_CRISTAL_MOL2):
            achou_linha = True
    return linha_info_cristal


def salva_molecula_em_arquivos_mol2_e_xyz(molecula, diretorio, nome_arquivo,
    ↪ linha_info_cristal):
    nome_arquivo_completo = os.path.join(diretorio, nome_arquivo)
    # Cria o arquivo mol2
    with io.MoleculeWriter(nome_arquivo_completo + ".mol2") as mol_writer:
        mol_writer.write(molecula)
    # Acrescenta as informações do cristal
    with open(nome_arquivo_completo + ".mol2", 'a') as arquivo_mol2:
```

```
        arquivo_mol2.write(MARCADOR_INFO_CRISTAL_MOL2 + "\n")
        arquivo_mol2.write(linha_info_cristal + "\n")

    # Cria o arquivo xyz
    with open(nome_arquivo_completo + ".xyz", 'w') as arquivo_xyz:
        # Número de átomos e descrição
        arquivo_xyz.write("%d\n%s\n" % (len(molecula.atoms), nome_arquivo))
        # Átomos e suas posições
        for atom in molecula.atoms:
            arquivo_xyz.write("{:4} {:17.12f} {:17.12f} {:17.12f}\n".format(
                atom.atomic_symbol, atom.coordinates.x, atom.coordinates.y, atom.
                    ↪ coordinates.z))


def mapeia_atomos_unidade_assimetrica_por_label(unidade_assimetrica):
    mapa = {}
    for atom in unidade_assimetrica.atoms:
        mapa[atom.label] = atom
    return mapa


def encontra_operadores(molecula, operadores_validos, mapa_unidade_assimetrica):
    operadores_string = ""
    ultimo_operador = None
    for atom in molecula.atoms:
        atomo_original = mapa_unidade_assimetrica[atom.label]
        # Depois que encontra um operador, é provável que os próximos átomos terão
            ↪ o mesmo operador
        if ultimo_operador:
            posicao_atomo_atual = np.array(atom.fractional_coordinates)
            if mesmo_ponto(aplica_operador(atomo_original, ultimo_operador),
                ↪ posicao_atomo_atual):
                continue
        operador_do_atomo = encontra_operador_do_atomo(operadores_validos, atom,
            ↪ atomo_original)
        ultimo_operador = operador_do_atomo

        if operadores_string:
            # Inclui separador entre os operadores da molécula
            operadores_string += " | "
        operadores_string += obtem_texto_operador(operador_do_atomo)

    return operadores_string


def encontra_operador_do_atomo(operadores_originais, atom, atomo_original):
    posicao_atomo_atual = np.array(atom.fractional_coordinates)
    for operador in operadores_originais:
        # Verifica se alguma translação deste operador resulta na posição do átomo
        posicao_apos_operador_inicial = aplica_operador(atomo_original, operador)
        if mesmo_ponto(posicao_atomo_normalizada(posicao_apos_operador_inicial),
                       posicao_atomo_normalizada(posicao_atomo_atual)):
            # Encontrado o operador original, corrige o vetor de translação
            diferenca_posicao = posicao_atomo_atual - posicao_apos_operador_inicial
            return {'rotacao': operador['rotacao'], 'translacao': operador['
                ↪ translacao'] + diferenca_posicao}


def obtem_texto_operador(operador):
```

```
    # Obtém vetor de multiplicadores de rotação a partir da matriz (soma da coluna)
    rotacao = np.sum(operador['rotacao'], axis=1)
    operador_str = ""
    variaveis = ["x", "y", "z"]
    for i in range(3):
        # A parte da translação só aparece se diferente de zero
        if not np.isclose(0.0, operador['translacao'][i]):
            translacao_str = str(round(operador['translacao'][i], 1))
            # Se inteiro, remove .0 do final
            if translacao_str.endswith(".0"):
                translacao_str = translacao_str[:-2]
            operador_str += translacao_str
            # Se a translação não foi zero, precisa do "+" após ela, se a rotação
                ↪ for positiva
            if rotacao[i] > 0:
                operador_str += "+"
        if abs(rotacao[i]) > 1:
            operador_str += rotacao[i]
        elif rotacao[i] == -1:
            operador_str += "-"
        operador_str += variaveis[i]
        if i < 2:
            operador_str += ","
    return operador_str


def main():

    # Seleção dos arquivos CIF
    root = tk.Tk()
    root.withdraw()
    string_arquivos = filedialog.askopenfilenames(filetypes=[("Arquivos CIF", "*.
        ↪ cif")])
    arquivos = root.tk.splitlist(string_arquivos)

    tempo_inicial_geral = time.time()

    for arquivo in arquivos:

        tempo_inicial = time.time()

        # Leitura do cristal do arquivo
        with io.CrystalReader(arquivo) as crystal_reader:
            crystal = crystal_reader[0]
            print("Feita a leitura do arquivo:", arquivo)

        print("Identificador do cristal:", crystal.identifier)
        print(crystal.cell_lengths)
        distancia_maxima_contato = max(crystal.cell_lengths) + 0.1

        # Obtém unidade assimétrica (conjunto de átomos definidos no arquivo CIF)
        unidade_assimetrica = crystal.asymmetric_unit_molecule

        linha_info_cristal = obtem_informacoes_do_cristal_para_mol2(crystal)

        operadores = []
        for symmop in crystal.symmetry_operators:
            matriz_rotacao = np.reshape(crystal.symmetry_rotation(symmop), (3, 3))
            operadores.append({'rotacao': matriz_rotacao,
```

```
                        'translacao': crystal.symmetry_translation(symmop)})

posicoes_normalizadas_invalidas = obtem_posicoes_normalizadas_invalidas(
    ↪ unidade_assimetrica, operadores)

mapa_unidade_assimetrica = mapeia_atomos_unidade_assimetrica_por_label(
    ↪ unidade_assimetrica)

# Obtém moléculas do cristal expandido:
# A cela unitária original vai de 0 a 1 (multiplicado pelos lados em
    ↪ Angstroms).
# Para garantir as moléculas completas circundantes, acrescento 2x esses
    ↪ tamanhos. Então fica de -2 a 3.
# Motivo: a M1 pode ser grande e estar apenas em parte na cela primária e o
    ↪  restante
# na(s) vizinha(s).
# A molécula vizinha dela pode fazer o mesmo, no máximo mais uma cela.
# A opção inclusion='AnyAtomIncluded' garante que só virão moléculas
    ↪ completas.

cristal_expandido = crystal.packing(box_dimensions=((-2, -2, -2), (3, 3, 3)
    ↪ ), inclusion='AnyAtomIncluded')
print("Concluída expansão do cristal")

# Lista de moléculas que podem ser a M1 (contêm átomos da unidade assimé
    ↪ trica)
candidatas_m1 = []

# Lista de átomos removidos devido a desordem
atomos_removidos_por_molecula = []

# Encontra a(s) molécula(s) (dentre as da expansão) que corresponde(m) à(s)
    ↪  da unidade assimétrica.
# Isso é necessário para que seja obtida a molécula completa, no caso de
    ↪ uma molécula aumentada por simetria.
# Assume-se que a molécula expandida contém todos os átomos de uma molécula
    ↪  da unidade assimétrica.
# Mas a expandida pode ser maior se átomos gerados por outro operador de
    ↪ simetria se unirem à molécula
# (como ocorre por exemplo no caso da molécula com simetria).
def encontra_candidatas_m1():
    moleculas_ua_restantes = unidade_assimetrica.components
    for molecula_unid_assimetrica in unidade_assimetrica.components:
        primeiro_atomo = molecula_unid_assimetrica.atoms[0]
        # Busca molécula que contém o primeiro átomo da molécula da unidade
            ↪  assimétrica
        loop_molecula_expansao_correspondente = \
            (molecula_expansao for molecula_expansao in cristal_expandido.
                ↪ components
             for atomo_expansao in molecula_expansao.atoms
             if primeiro_atomo.label == atomo_expansao.label
             and mesmo_ponto(primeiro_atomo.coordinates, atomo_expansao.
                 ↪ coordinates))
        for molecula_expansao in loop_molecula_expansao_correspondente:
            # Encontrou molécula correspondente à da unidade assimétrica -
                ↪ candidata à M1.
            # Evita adicionar duas vezes a mesma molécula (em situações
                ↪ excepcionais)
            if molecula_expansao not in candidatas_m1:
```

```
                    if len(molecula_expansao.atoms) < len(
                        ↪ molecula_unid_assimetrica.atoms):
                        raise ValueError("Molécula expandida tem menos átomos
                            ↪ que a correspondente "
                                         "da unidade assimétrica.")
                    if len(molecula_expansao.atoms) > len(
                        ↪ molecula_unid_assimetrica.atoms):
                        # Remove os átomos inválidos da molécula (se houver) e
                            ↪ guarda quais foram os removidos
                        atomos_removidos_m1 = remove_atomos_invalidos(
                            molecula_expansao, posicoes_normalizadas_invalidas)
                        if atomos_removidos_m1:
                            atomos_removidos_por_molecula.append(
                                {'molecula': molecula_expansao, 'atomos':
                                    ↪ atomos_removidos_m1})
                    # Adiciona a molécula às candidatas à M1
                    candidatas_m1.append(molecula_expansao)
                # Termina quando todas as moléculas da unidade assimétrica
                    ↪ foram encontradas
                moleculas_ua_restantes.remove(molecula_unid_assimetrica)
                if not moleculas_ua_restantes:
                    return
encontra_candidatas_m1()

sufixo_m1 = 'A'
if len(candidatas_m1) > 1:
    print("Encontradas", len(candidatas_m1), "moléculas candidatas a M1.")
# Faz os cálculos para cada uma das candidatas a M1
for posicao_m1 in range(len(candidatas_m1)):
    # Define nome da M1 atual
    nome_m1 = "M1"
    if len(candidatas_m1) > 1:
        nome_m1 += sufixo_m1
        print("Processando", nome_m1 + "...")

    molecula_m1 = candidatas_m1[posicao_m1]

    # Identificação das áreas de contato entre as moléculas (geração do
        ↪ poliedro de Voronoi-Dirichlet)

    # vetor de posicoes de todos os átomos a serem considerados no poliedro
    posicoes_atomos = []
    molecula_por_posicao = []

    # guarda a menor e maior coordenada, para usar como limite no cálculo
        ↪ do poliedro
    menor_xyz = [0.0, 0.0, 0.0]
    maior_xyz = [0.0, 0.0, 0.0]

    # inclui primeiro os átomos da M1
    inclui_posicoes_atomos_da_molecula(molecula_m1, posicoes_atomos,
        ↪ molecula_por_posicao, maior_xyz,
                                       menor_xyz)

    passou_m1 = False
    pontos_m1 = [atom.coordinates for atom in molecula_m1.atoms]
    conta_moleculas = 0
    # varre todas as moléculas e adiciona cada uma que tiver átomo próximo
    # da M1 (até distancia_maxima_contato)
```

```
print("Mapeando moléculas com átomos a até", round(
    ↪ distancia_maxima_contato, 6), "Å de átomos da M1...")
for molecula in cristal_expandido.components:
    # verifica se é a própria M1 (enquanto não passar dela)
    # quando passar dela, não precisa mais verificar
    if not passou_m1:
        if possui_ponto(molecula.atoms[0].coordinates, pontos_m1):
            # pula M1 (já adicionada)
            passou_m1 = True
            continue

    if posicoes_normalizadas_invalidas.values:
        # Remove os átomos inválidos da molécula e guarda quais foram
            ↪ os removidos
        atomos_removidos = remove_atomos_invalidos(molecula,
            ↪ posicoes_normalizadas_invalidas)
        if atomos_removidos:
            atomos_removidos_por_molecula.append({'molecula': molecula,
                                                  'atomos':
                                                      ↪ atomos_removidos
                                                      ↪ })

    class ContatoEncontrado(Exception):
        pass
    # verifica se algum átomo está próximo à M1
    try:
        for atom in molecula.atoms:
            for atom_m1 in molecula_m1.atoms:
                if distancia_pontos(atom.coordinates, atom_m1.
                    ↪ coordinates) < distancia_maxima_contato:
                    raise ContatoEncontrado
    except ContatoEncontrado:
        conta_moleculas += 1
        inclui_posicoes_atomos_da_molecula(molecula, posicoes_atomos,
            ↪ molecula_por_posicao, maior_xyz,
                                           menor_xyz)
print(conta_moleculas, "moléculas encontradas no distanciamento limite.
    ↪ ")

# Calcula poliedro de Voronoi conforme limites observados anteriormente
voronoi = pyvoro.compute_voronoi(
    posicoes_atomos, [[menor_xyz[0]-0.5, maior_xyz[0]+0.5], [menor_xyz
        ↪ [1]-0.5, maior_xyz[1]+0.5],
                      [menor_xyz[2]-0.5, maior_xyz[2]+0.5]],
                          ↪ distancia_maxima_contato)
print("Preparado poliedro de Voronoi")

# Mapeia átomos de outras moléculas em contato com a M1.
moleculas_contato = []
area_contato_por_molecula = []
# Os primeiros átomos são da M1: olha átomo por átomo dela
for i in range(len(molecula_m1.atoms)):
    faces_contato = voronoi[i]['faces']
    for face in faces_contato:
        pos = face['adjacent_cell']
        # verifica somente os contatos com outras moléculas
        # (pula os da própria M1 e os negativos (que incluem fronteira)
            ↪ )
        if pos >= len(molecula_m1.atoms):
```

```
                molecula_contato = molecula_por_posicao[pos]
                vertices = []
                for posVertice in face['vertices']:
                    vertices.append(voronoi[i]['vertices'][posVertice])
                if molecula_contato not in moleculas_contato:
                    moleculas_contato.append(molecula_contato)
                    area_contato_por_molecula.append(
                        {'molecula': molecula_contato, 'area': calcula_area
                            ↪ (vertices)})
                else:
                    for area_molecula in area_contato_por_molecula:
                        if area_molecula['molecula'] == molecula_contato:
                            area_molecula['area'] += calcula_area(vertices)
                            break

print("Finalizada identificação dos contatos e suas áreas")
print("Total de moléculas em contato encontradas:", len(
    ↪ moleculas_contato))

diretorio = os.path.dirname(arquivo)
nome_arquivo = os.path.basename(arquivo).replace(".", "_")
sufixo_nome_arquivo = ""
if len(candidatas_m1) > 1:
    sufixo_nome_arquivo = "_" + nome_m1
dir_monomeros = "monomeros_" + nome_arquivo + sufixo_nome_arquivo
os.makedirs(os.path.join(diretorio, dir_monomeros), exist_ok=True)
dir_dimeros = "dimeros_" + nome_arquivo + sufixo_nome_arquivo
os.makedirs(os.path.join(diretorio, dir_dimeros), exist_ok=True)
numeracao_molecula = 1

salva_molecula_em_arquivos_mol2_e_xyz(molecula_m1, os.path.join(
    ↪ diretorio, dir_monomeros),
                                      nome_arquivo + "_" + nome_m1,
                                          ↪ linha_info_cristal)

# Objeto que vai incluir a M1 e as demais moléculas do cluster
cluster = molecula_m1.copy()

conteudo_csv = [["Symmetry Code", "Dimer", "Area"]]

area_total = sum([area_molecula['area'] for area_molecula in
    ↪ area_contato_por_molecula])
operadores_molecula = encontra_operadores(molecula_m1, operadores,
    ↪ mapa_unidade_assimetrica)
conteudo_csv.append([operadores_molecula, nome_m1, round(area_total, 4)
    ↪ ])

moleculas_cluster = [{'molecula': molecula_m1, 'id': nome_m1}]

print("Preparando arquivos")
for area_molecula in area_contato_por_molecula:
    molecula = area_molecula['molecula']
    numeracao_molecula += 1
    id_molecula = "M" + str(numeracao_molecula)
    moleculas_cluster.append({'molecula': molecula, 'id': id_molecula})
    area = area_molecula['area']

    # Salva arquivos xyz e mol2: um com a molécula, outro com o dímero
        ↪ M1 + molécula atual
```

```
            salva_molecula_em_arquivos_mol2_e_xyz(molecula, os.path.join(
                ↪ diretorio, dir_monomeros),
                                                  nome_arquivo + "_" +
                                                      ↪ id_molecula,
                                                      ↪ linha_info_cristal)

            dimero = molecula_m1.copy()
            dimero.add_molecule(molecula)
            salva_molecula_em_arquivos_mol2_e_xyz(
                dimero, os.path.join(diretorio, dir_dimeros),
                nome_arquivo + "_" + nome_m1 + "_" + id_molecula,
                    ↪ linha_info_cristal)

            cluster.add_molecule(molecula)

            operadores_molecula = encontra_operadores(molecula, operadores,
                ↪ mapa_unidade_assimetrica)

            conteudo_csv.append([operadores_molecula, nome_m1 + "---" +
                ↪ id_molecula, round(area, 4)])

        salva_molecula_em_arquivos_mol2_e_xyz(cluster, diretorio, nome_arquivo
            ↪ + "_cluster" + sufixo_nome_arquivo,
                                              linha_info_cristal)

        conteudo_csv.append(["N:␣" + str(len(moleculas_contato)), "Total:",
            ↪ round(area_total, 4)])

        # Acrescenta coordenadas da M1
        conteudo_csv.append([])
        conteudo_csv.append(["Atoms␣from␣" + nome_m1 + ":"])
        conteudo_csv.append(["Atom␣Label", "Fractional␣Coordinates", "Symmetry␣
            ↪ Code"])
        for atom in molecula_m1.atoms:
            atomo_original = mapa_unidade_assimetrica[atom.label]
            operador_do_atomo = encontra_operador_do_atomo(operadores, atom,
                ↪ atomo_original)
            coords = atom.fractional_coordinates
            conteudo_csv.append([atom.label, "x=" + str(round(coords.x, 4)) + "
                ↪ ,␣y=" + str(round(coords.y, 4))
                                 + ",␣z=" + str(round(coords.z, 4)),
                                     ↪ obtem_texto_operador(operador_do_atomo)
                                     ↪ ])

        nome_arquivo_completo = os.path.join(diretorio, nome_arquivo +
            ↪ sufixo_nome_arquivo)
        with open(nome_arquivo_completo + ".csv", 'w', newline='') as
            ↪ arquivo_csv:
            writer = csv.writer(arquivo_csv, delimiter=';')
            writer.writerows(conteudo_csv)

        print("Arquivos␣gerados␣na␣pasta:␣" + diretorio)

        # Atualiza o sufixo para a próxima M1, se houver: obtém próxima letra
        sufixo_m1 = chr(ord(sufixo_m1) + 1)
    print("Tempo␣decorrido␣para␣o␣cristal:", round(time.time() - tempo_inicial,
        ↪ 2), "s")
```

```
    print("Tempo total decorrido:", round(time.time() - tempo_inicial_geral, 2), "s
        ↪ ")
    print("Processamento concluído")


# Execução do método principal
main()
```

Apêndice B

Arquivos de saída

A seguir são listados os conteúdos de alguns dos arquivos de saída do *software*, resultantes da execução para o cristal com código CCDC LERKIE.

Planilha: arquivo: LERKIE_cif.csv

Symmetry Code	Dimer	Area
x,y,z	M1	268.4896
2-x,1-y,-z	M1—M2	36.9773
1-x,1-y,-z	M1—M3	28.1463
x,-1+y,z	M1—M4	7.7676
1+x,y,z	M1—M5	46.3574
2-x,-y,-z	M1—M6	2.039
1+x,-1+y,z	M1—M7	5.0661
-1+x,y,z	M1—M8	46.3574
1.5-x,-0.5+y,0.5-z	M1—M9	18.7686
0.5-x,-0.5+y,0.5-z	M1—M10	22.0568
1.5-x,0.5+y,0.5-z	M1—M11	18.7686
0.5-x,0.5+y,0.5-z	M1—M12	22.0568
-1+x,1+y,z	M1—M13	5.0661
x,1+y,z	M1—M14	7.7676
1-x,2-y,-z	M1—M15	1.2938
N: 14	Total:	268.4896
Atoms from M1:		
Atom Label	Fractional Coordinates	Symmetry Code
Cl1	x=0.9928, y=0.1086, z=0.0733	x,y,z
O1	x=0.592, y=0.6502, z=0.1167	x,y,z
N1	x=0.4553, y=0.3701, z=0.2368	x,y,z

C1	x=0.356, y=0.4683, z=0.2547	x,y,z
H1	x=0.2535, y=0.4746, z=0.2956	x,y,z
C2	x=0.3905, y=0.5687, z=0.2173	x,y,z
H2	x=0.3127, y=0.6384, z=0.2333	x,y,z
C3	x=0.54, y=0.5615, z=0.1571	x,y,z
C4	x=0.647, y=0.4548, z=0.1354	x,y,z
C5	x=0.7964, y=0.4349, z=0.0751	x,y,z
H3	x=0.8253, y=0.496, z=0.0467	x,y,z
C6	x=0.901, y=0.3308, z=0.056	x,y,z
H4	x=1.0014, y=0.3204, z=0.0151	x,y,z
C7	x=0.8573, y=0.2393, z=0.0979	x,y,z
C8	x=0.7075, y=0.2521, z=0.1561	x,y,z
H5	x=0.6752, y=0.1889, z=0.1829	x,y,z
C9	x=0.599, y=0.3593, z=0.1771	x,y,z
C10	x=0.485, y=0.7596, z=0.1381	x,y,z
H6	x=0.6133, y=0.7819, z=0.1771	x,y,z
H7	x=0.2391, y=0.7595, z=0.148	x,y,z
C11	x=0.5575, y=0.8378, z=0.0839	x,y,z
H8	x=0.5083, y=0.9145, z=0.0972	x,y,z
H9	x=0.4143, y=0.8184, z=0.0469	x,y,z
H10	x=0.7976, y=0.8318, z=0.0722	x,y,z

Arquivo XYZ da M1: LERKIE_cif_M1.xyz

```
24
LERKIE_cif_M1
Cl      3.888745259343      1.301028000000      1.528964967593
O       2.303951304848      7.789396000000      2.434245726031
N       1.741961020634      4.433798000000      4.939412064475
C       1.347863984102      5.610234000000      5.312788229821
H       0.936495327447      5.685708000000      6.165921479133
C       1.491143063783      6.813026000000      4.532661493287
H       1.181971397609      7.648032000000      4.866405551697
C       2.091250587760      6.726770000000      3.276949473518
C       2.516398765008      5.448504000000      2.824309094298
C       3.116133476899      5.210102000000      1.566511174164
H       3.235558444356      5.942080000000      0.974115470486
C       3.531314768393      3.962984000000      1.168104204437
H       3.934426225049      3.838392000000      0.314970955125
```

C	3.350943832602	2.866814000000	2.042096457399
C	2.750064466896	3.020158000000	3.256090469867
H	2.617596906056	2.263022000000	3.815111767705
C	2.319161005043	4.304414000000	3.694129546531
C	1.878864291341	9.100008000000	2.880628404155
H	2.375388605043	9.367162000000	3.694129546531
H	0.909967087896	9.098810000000	3.087132540297
C	2.174984540503	10.036844000000	1.750070406290
H	1.978818547997	10.955710000000	2.027495154844
H	1.619465668529	9.804432000000	0.978287271216
H	3.121443126393	9.964964000000	1.506020063577

Arquivo XYZ da M2: LERKIE_cif_M2.xyz

24			
LERKIE_cif_M2			
Cl	3.975254740657	10.678972000000	−1.528964967593
O	5.560048695152	4.190604000000	−2.434245726031
N	6.122038979366	7.546202000000	−4.939412064475
C	6.516136015898	6.369766000000	−5.312788229821
H	6.927504672553	6.294292000000	−6.165921479133
C	6.372856936217	5.166974000000	−4.532661493287
H	6.682028602391	4.331968000000	−4.866405551697
C	5.772749412240	5.253230000000	−3.276949473518
C	5.347601234992	6.531496000000	−2.824309094298
C	4.747866523101	6.769898000000	−1.566511174164
H	4.628441555644	6.037920000000	−0.974115470486
C	4.332685231607	8.017016000000	−1.168104204437
H	3.929573774951	8.141608000000	−0.314970955125
C	4.513056167398	9.113186000000	−2.042096457399
C	5.113935533104	8.959842000000	−3.256090469867
H	5.246403093944	9.716978000000	−3.815111767705
C	5.544838994957	7.675586000000	−3.694129546531
C	5.985135708659	2.879992000000	−2.880628404155
H	5.488611394957	2.612838000000	−3.694129546531
H	6.954032912104	2.881190000000	−3.087132540297
C	5.689015459497	1.943156000000	−1.750070406290
H	5.885181452003	1.024290000000	−2.027495154844
H	6.244534331471	2.175568000000	−0.978287271216
H	4.742556873607	2.015036000000	−1.506020063577