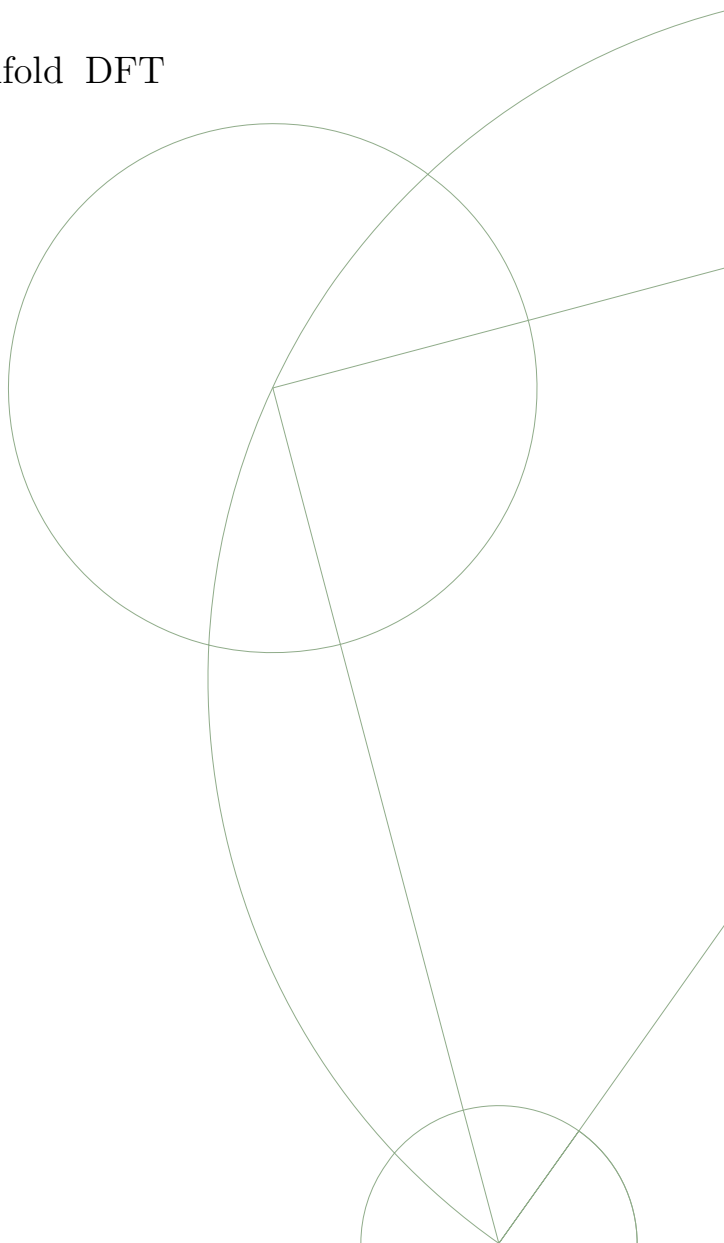
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

M.Sc. in Physics

David Dedenbach mnk942@alumni.ku.dk

Combinatorial Geometry of Fullerenes

Towards a Mixed-dimensional Carbon Manifold DFT



Supervised by Assoc. Prof., Ph.D. James E. Avery

August 16, 2021

Abstract

The amount of distinct Fullerene molecules, a class of polyhedral, hollow carbon molecules with spherical topology and graphene-like structure, is enormous. The quest to find synthesizable isomers with useful properties requires fast and effective solutions for approximation. The CARMA project, which this thesis is a part of, tries to achieve that by describing the electronic structure of the molecules with a two-dimensional, coordinate free surface DFT restricted to the molecule shells by means of a graph theory-based description. This approach hopes to unify the good accuracy of a classical, three-dimensional DFT approach with the speed benefits of other graph-theoretical approaches. Reducing the dimension yields losses in accuracy, no matter how good the DFT works on the surface, the reason being that high curvature regions of the molecule can yield out-of-surface interactions of high strength due to short distances between more or less opposing regions of the shell.

In this thesis project, a mixed-dimensional treatment is proposed to compute relevant out-of-surface interactions by embedding high-curvature regions of the surface into three-dimensional space. The approach aims to reach high computation speeds by embedding only the necessary regions, based on the molecule's graph bonds and without taking into account further physical restrictions.

Methods are developed and implemented in software to structure these embeddings and obtain valid geometries for any possible high-curvature region. The current state of the implementation is discussed, including descriptions of the problems that are still to solve. An overview is given about what work lies ahead, in terms of concept development, improving the current implementation and also continuing development towards the construction of molecule regions of arbitrary size.

Contents

1. Introduction

Carbon is one of the most variable elements in existance. There are countless ways in which it can form molecules, in combination with other elements, but also purely with carbon atoms. When thinking about molecules made purely of carbon, what comes to mind first is diamond. For some, it is also graphene or carbon nanotubes, as these two structures have seen a fair bit of research in the past decades, due to their excellent physical properties. Both graphene and carbon nanotubes have a hexagonal structure, where each carbon atom is bonded with three other atoms. While graphene is one big, flat sheet made of hexagons, nanotubes have a cylindrical shape, in a way they are graphene curled up to a cylinder. Relaxing the the structural requirements a little bit and introducing some pentagons to the hexagonal structure, one can now form surfaces with (positive) intrinsic curvature. This leads to the molecule class of *Fullerenes*. Fullerenes consist of exactly twelve faces of carbon pentagons, around these pentagons more hexagonal carbon faces are arranged. The exact number of twelve pentagons gives the molecules a *spherical topology*. In one way or the other, Fullerenes always resemble hollow balls of carbon, although their shapes can vary greatly. The easiest example of a Fullerene is the so-called *Buckyball*, chemically the molecule is referred to as $C_{60}-I_h$. It is sometimes also called the "football molecule", because it has the same structure as a football.

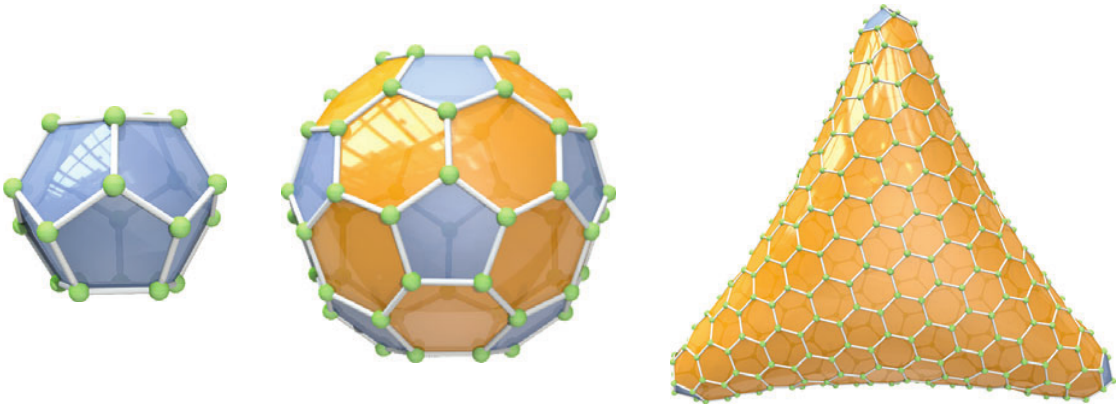


Figure 1.1.: Some Fullerenes, from left to right: The smallest Fullerene $C_{20}-I_h$, the $C_{60}-I_h$ *Buckyball* and the $C_{440}-D_3$ with trihedral shape. From [13] with permission.

But also more complicated shapes are possible, for example the trihedral $C_{440}-D_3$ shown in Figure 1.1. Fullerenes have several prospective applications, which make more extensive

research on the whole molecule class worth it. For example, they are currently used in organic solar cells to increase the material conductivity [6], it is projected that they can be used to store hydrogen at near-metallic density [12]. Other possible or real applications include organic electronics [7], printable electronics [4] and asthma inhibitors [10].

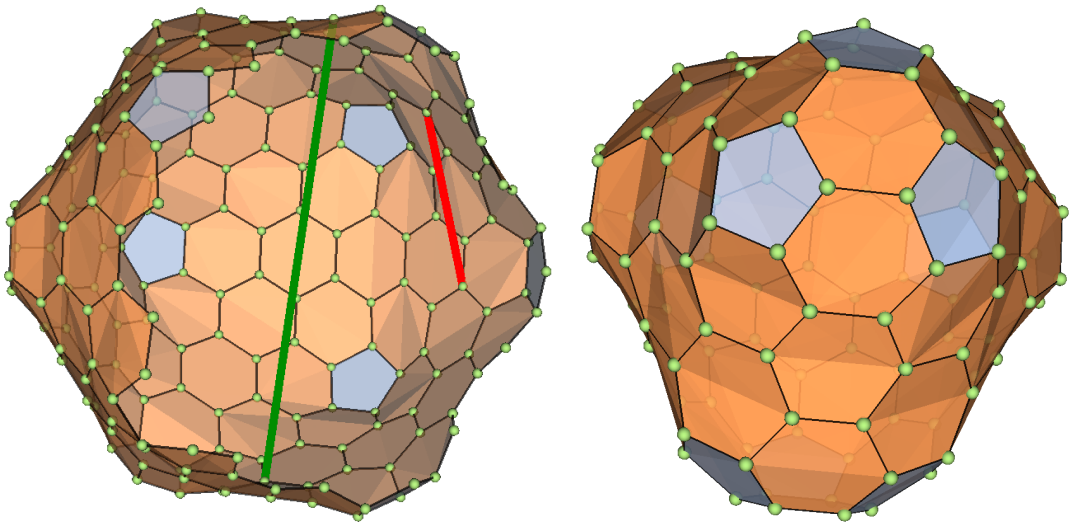
This thesis is part of a research project called CARMA, which is aimed at predicting the properties of Fullerene molecules, finding suitable candidates for synthesis and finding ways to synthesize them. CARMA is short for CARbon MAnifolds and reflects the approach that is taken in this project. The goal is to describe Fullerenes as a two-dimensional *surface manifold*, which does not require a three-dimensional embedding of the molecule. The existing methods for determining Fullerene properties are mostly based on quantum chemical computations, where the full three-dimensional structure of nuclei and electronic bonds is computed. This is very time consuming though, generally supercomputers are needed to cut the simulation of a single molecule down to a reasonable time span. Even with a supercomputer, computing the properties of a single molecule can take days, depending on the level of detail that is applied.

This is in theory not a big problem when simulating a single molecule. The problem is to find, which molecules are worth simulating on a high level of detail. The number of Fullerene isomers for a given number of carbon atoms N scales in the order of $\mathcal{O}(N^9)$. The smallest Fullerene is C_{20}, and given that the number of carbon atoms goes to the hundreds quite quickly, the number of isomers reaches the millions very fast. For example, there exist 285914 isomers for $N = 100$ and already 214127742 for $N = 200$ [3]. These numbers are huge, and the question arises, what isomers to select for a more thorough investigation. A set of methods is needed, which roughly approximate the properties of a single isomer very quickly, such that many isomers can be run through the method quickly and the most promising ones for a certain purpose can be filtered out. For Fullerenes, one approach can be to reduce the dimensionality of the describing theory: Fullerenes are hollow shells, one can assume that the interior of the molecule does not contribute a lot to the molecular properties compared to the in-surface dynamics. Electronic interactions will always mainly take place along the surface, as the Coulomb force scales with $1/r^2$ with distance. Additionally, electrons can move quite well through the surface - in graphene-like structures, the p-orbitals perpendicular to the surface connect to form a π-band, which enhances the conductivity along the surface. Electron-Electron interactions along the surface are not met by a lot of resistance. Overall, this invites to explore methods that only include electron dynamics in the molecule surface, instead of describing it in the full 3-dimensional molecule. However, just reducing the electron dynamics to the molecule shell does not provide enough of a speed boost to compute molecular properties in such a huge isomer space. A big speed problem is finding an approximation of the molecule shape itself. A method needs to be developed that drops the need for three-dimensional coordinate-based description of Fullerenes.

The CARMA project is working on providing such a method. The approach is based on a graph-theoretical description of Fullerenes. A graph consists of edges and vertices, which is a good fit to represent the atoms and bonds of a Fullerene. This graph structure forms

a two-dimensional *discrete surface manifold*, which can be used for electronic structure approximation. The most common tool for the approximation of electronic multi-body problems is a *density functional theory* (DFT). A part of the CARMA project is to find a DFT that is tailored for discrete non-Euclidean two-dimensional manifolds like the Fullerene surface manifolds. These manifolds are coordinate-free and provide a truly two-dimensional description of Fullerene molecules.

However, no matter if there is a coordinate free two-dimensional description or if the surface manifold is still embedded into three-dimensional space, reducing electron dynamics to the surface means ignoring the electronic interactions perpendicular to the surface. If the molecule was planar like graphene, those would not be of relevance. Also in Fullerenes, the approximation is in general still good, because many out-of-surface interactions are over such long distances that they become irrelevant. A good example



(a) View into $C_{240}-I_h$. Short and strong (red) versus long and weak (green) interaction.

(b) The $C_{120}-T$ has pockets, where also shorter interactions can be out-of-surface.

Figure 1.2.: Generally, the two-dimensional description is a good approximation, because shorter interactions are well-captured by in-surface approximations, while longer ones are too weak to be relevant (a). However, if a molecule has "pocket regions", there can be relevant out-of-surface interactions (b).

for this is shown in Figure 1.2a, where the green line marks an interaction between electrons on opposite sides of the hull, where the red line represents a much shorter interaction. The green interaction is clearly out-of-surface and can not be represented correctly in a surface DFT. But it is also weak, because it is very long-distance and therefore not of great importance to the overall accuracy of the approximation. On the other hand, the red interaction is much shorter and stronger, but also well-approximated by the surface. The fact that the approximation of interactions becomes better the

shorter the interactions are, combined with the distance scaling of $1/r^2$ in the Coulomb force makes the surface approximation more accurate, the bigger the Fullerene is, because the interaction strengths of out-of-surface interactions steadily decrease with molecule size.

This all sounds very promising already, yet there is another factor to be taken into account when it comes to how good a surface DFT can approximate the electronic structure. This factor is the distribution of curvature in the molecule surface. The icosahedral C_{240} in Figure 1.2a has the most even distribution of curvature that is possible. That is not always the case though, Figure 1.2b shows a $C_{120}-T$ with several high-curvature regions for direct comparison. In these cases, there can be situations with significant out-of-surface interactions.

Significant Out-Of-Surface Interactions

These situations can arise two different ways: For one, they can be based on the overall shape of the molecule. Two examples for that would be carbon-nanotube-like molecules or very flat Fullerenes. Nanotube-shaped Fullerenes have very little distances between between opposite surface walls and very flat Fullerenes are subject to the same, although in a slightly less regular manner. Both cases likely need a three-dimensional treatment or some other kind of specialized model anyway. Luckily for us, who are searching through the isomer space of Fullerenes for useful molecules, these cases do not appear frequently and can be awarded a special treatment.

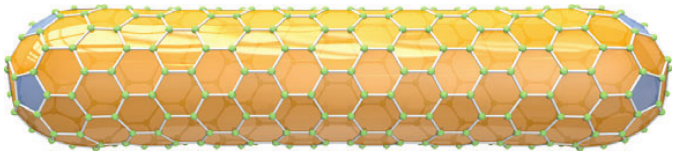
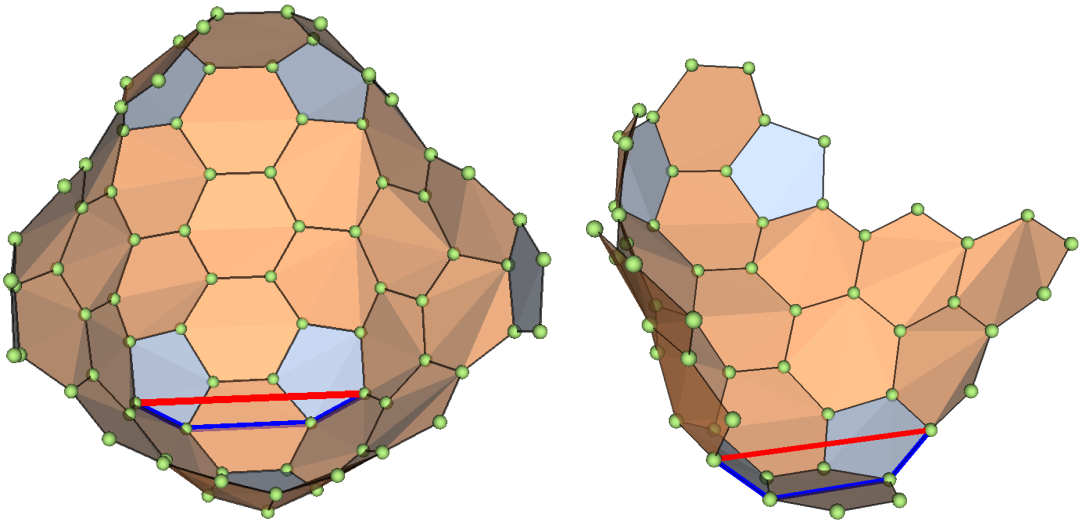


Figure 1.3.: The $C_{360}-D_{5h}$ with a nanotube-like shape, its out-of-surface interactions appear all over the molecule, surface DFT is not likely to be very accurate. From [13] with permission.

The far more common case of relevant out-of-surface interactions arises from the curvature of the molecule surface. For the great majority of isomers, the pentagons that are sources of curvature, are not equally distributed over the molecule surface. In fact, in many cases, they tend to group up tightly and build little ”pockets” in the molecule. No matter the size of the molecule, there can never be more than twelve of those regions, restricted by the fixed number of pentagon faces in the Fullerene. One could argue that in the most important cases, it is likely that there are less than twelve, because a deep pocket is likely to be shaped by more than one pentagon. To achieve a good accuracy in electronic structure computations, which ultimately determines the ability to project molecule properties, the out-of-surface interactions in the molecule’s pockets need to be included.

Very simplified, Figure 1.2 shows a short-distance out-of-surface interaction in a pocket of the molecule surface, the direct three-dimensional path in red and the in-surface interaction in blue.



(a) (b)

Figure 1.4.: View into $C_{120}-T$, where a short-distance interaction (red) is happens in a pocket of the molecule (a). Compared to its blue in-surface equivalent, this interaction is clearly out-of surface (b) and can not be approximated efficiently with a surface DFT.

The red interaction can not be approximated efficiently by a purely two-dimensional surface DFT, because the equivalent interaction through the surface is a lot longer in comparison. This is a short-distance interaction though and it has a higher strength than the out-of-surface interaction in Figure 1.2a, such that it can not just be omitted. Exactly that problem is what this thesis seeks to find a solution to.

Including Short-Distance Out-Of-Surface Interactions

The ansatz for approaching the problem of out-of-surface interactions in pocket regions of the molecule is a mixed-dimensional one. That is, to construct embeddings of the pockets in three-dimensional space up to some size that is adequate to cut off the approximation of out-of-surface interactions. Partly returning to a three-dimensional description does of course yield losses in computation speed, yet the fact that there can at max be twelve pockets means that these losses do not scale with the size of the molecule, i.e. the computation speed still scales with the two-dimensional description of the surface instead of the full three-dimensional simulation. Additionally, because of the inverse squared scaling of the Coulomb-force with distance, the size of the pocket region embeddings

does not necessarily need to scale with the size of the molecule, as the size of a pocket region can be chosen up to some cutoff-distance determined by the coulomb-interaction strength, as long as it includes all the curvature-generating pentagon faces.

Making such an approach a reality first and foremost requires a construction method for the pockets that satisfies certain conditions. To fit into the model, the construction should be based on the intrinsic geometry the surface manifold. It needs to be able to grow the embeddings of pockets to an arbitrary size that has been identified as the cutoff size. This growing process should be somewhat symmetrical, such that the embedding has the shape of a cone with some kind of even "radius", such that the cutoff is at similar interaction strengths into every direction. The term cone emphasizes the fact that upon growing the pocket regions, the specific arrangement of the pentagons in the center of the cone becomes less important. On a larger scale, one can assume point curvature in the center of the cone and a flat geometry otherwise. An important detail is that single regions have to be embedded without computing the embedding of the full molecule, because this would equate to a three-dimensional scaling of computation time, the very thing that we are trying to avoid with this method. This is a crucial point that eliminates existing methods for Fullerene shape approximation from contention and requires the development of a new one.

In summary, this thesis introduces a method for the construction of pocket region embeddings of Fullerene molecules, based on the intrinsic geometry information from the bond graph of the two-dimensional surface manifold. For a regional embedding, no information about the three-dimensional shape of the molecule is needed, and the goal is that the embedding can be grown to an arbitrary desired size.

1.1. Organization of This Thesis

The remainder of this thesis is organized as follows: Chapter 3 and 3 address already concepts for the theoretical description of Fullerenes. Both of these chapters do not address research made in the course of this thesis, but build a theoretical background to dive into the project's content.

Chapter marks the beginning of my personal work, which fills all chapters from this point. Here, some additional structure is introduced that is needed to describe the embeddings of surface regions and their construction. Chapter 5 transfers this structure to a concept for an algorithm to build the embeddings. The chapters 8, 9 and 10 describe the implementation of this algorithm, where the first two chapters are very implementation-heavy and 10 is a little more focused on concepts in development towards the end stages of the construction process, which yield problems that are not solved yet or only partially solved. Finally, chapter 11 gives an overview of the capabilities that the algorithm has in it's current state, and where there are problems that need to be solved.

1.2. Contributions of this Thesis to CARMA

This thesis has given the following contributions to the CARMA project:

1. Development of a three-step method to embed surface manifold regions of Fullerenes into three-dimensional space (chapters 4 and 5)
2. Building the first iteration of an implementation, mainly for step one (chapter 8) and two (chapter 9). Starting to build an implementation for the last step (chapter 10).

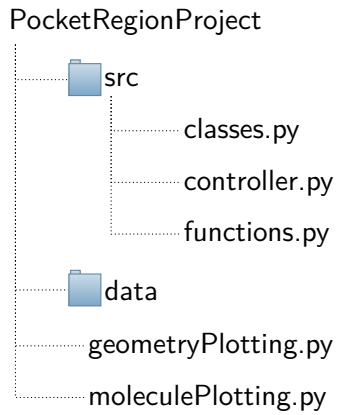
The contents of chapter 2 and chapter 3 are background information and context to make this thesis more understandable. Parts of these chapters are research done by the CARMA group, parts are general research about Fullerenes. All contents of the chapters 4 to 11 describe original work by myself.

1.3. Software

All of the software that is relevant for this thesis can be found on the

folding-carbon github

under the folder PocketRegionProject. The file structure is shown below.



For most of this thesis, the most important folder is src. The file functions.py holds all basic functions that are used globally. This file will be referred to several times during the thesis. controller.py holds the `constructionManager` class, this will be important in chapter 6 and classes.py holds the `Patch` and `PocketRegion` classes, which will be important in chapters 6, 8 and 9.

The folder data holds Fullerene geometries that serve as source for the files geometryPlotting.py and moleculePlotting.py, where one can plot three-dimensional, interactive plots of Fullerenes, pocket regions etc.

The interested reader is invited to come back to this chapter before searching for functions that are referenced in the text. For important functions, there will be a reference on where to find them in the file structure.

Background

2. Fullerenes

In this first chapter, I want to dive into the mathematical description of Fullerenes that the CARMA project is using. I have stated that Fullerenes are described as graphs, where the carbon atoms are represented by vertices and the chemical bonds by edges. I will introduce the mathematical framework of graph theory and Fullerene graphs. There are two different ways of representing Fullerenes through graphs, I will introduce both, address advantages and disadvantages and afterwards talk about the numerical representation of graphs.

2.1. Combinatorial Geometry of Fullerenes

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a set of vertices (also called nodes) $\mathcal{V}$, which are connected by a set of edges $\mathcal{E}$. Mathematically, the set $\mathcal{E}$ can be described as a subset of the set of all possible pairs of vertices, given that only distinct vertices can be paired; $\mathcal{E} \subseteq \{(x, y) | (x, y) \in \mathcal{V}^2 \wedge x \neq y\}$.

Fullerene graphs are a type of graphs that is called *polyhedral* graphs, fulfilling two crucial properties. Polyhedral graphs are *planar*, which means that they can be embedded in two-dimensional Euclidean space with a map $\mathcal{V} \to \mathbb{R}^2$, such that all edges can be drawn straight without crossing. Most importantly, a planar graph can also be embedded on a two-dimensional sphere, which makes a lot of sense for Fullerenes with a spherical topology. Furthermore, polyhedral graphs are *three-connected*; at any point in the graph, three edges or more would have to be removed to create two disconnected graphs. Also this makes sense, as every carbon atom in a Fullerene has three bonds with other atoms. The combination of these two properties yiels a graph that has a *unique* embedding in three dimensions, of course still with a spherical topology [13].

It is very intuitive to represent carbon atoms by nodes and their bonds by edges in a graph. This representation, where nodes represent carbon molecules and edges represent bonds, is characterized by each node having degree three. A graph with that property is called a *cubic* graph, which is why this representation is called the *cubic representation*. However, this graph can be impractical to work with, for example it is rather hard to represent the graph in R^2. A later chapter will be dedicated to addressing some problems that arise when working with this representation in three-dimensional embeddings of the molecule. To provide an alternative to the cubic representation, the *dual representation* will now be introduced.

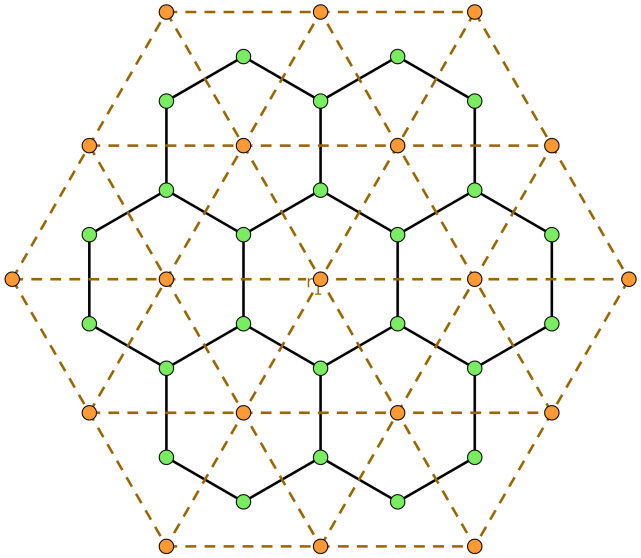


Figure 2.1.: Cubic and dual representation of a standard hexagon grid. Cubic nodes in black and edges in blue, dual nodes in orange and edges in black.

The dual representation of a Fullerene molecule can be constructed by placing a node in the center of each face and then connecting nodes that represent adjacent faces. A simple example in the two-dimensional plane is shown in figure 2.1. The green nodes represent the vertices of the cubic representation, the black edges represent the bonds. Placing a dual vertex in the middle of each cubic face (and continuing the grid a little further for illustration purposes) yields the orange nodes, connected by dotted brown edges. The edges of the dual representation represent two adjacent faces in the cubic representation and cross the common edge of those two faces in the two-dimensional picture. Dualizing a representation works in both directions. The dual of the dual

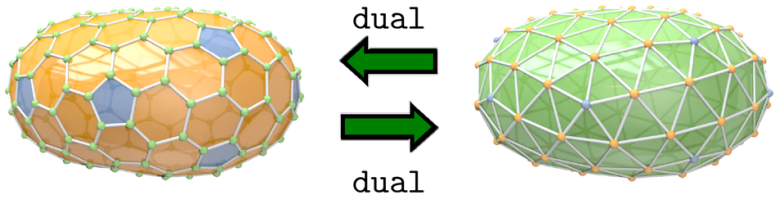


Figure 2.2.: Cubic and dual representation of a $D_{3h}-C_{224}$ molecule. From [13] with permission.

representation yields the cubic representation. This immediately becomes clear when taking another look at Figure 2.1: Each of the green nodes is in the center of a dual face

and a black edge connects two dual triangles.

Figure 2.2 shows the cubic and dual representation of a whole molecule. The color coding in this figure is noteworthy: the cubic representation consists of green nodes, orange hexagon faces and blue pentagon faces. Equivalently, the dual representation has orange 'hexagon nodes' and blue 'pentagon nodes', as well as green faces. This color coding will be kept as far as possible during this thesis. Also, I want to make it clear that Figure 2.2 shows the *embeddings* of Fullerene surface manifolds, where the edge lengths have been optimized such that the shape resembles the physical shape of the molecule. This is not what the electronic structure is actually calculated on, otherwise every surface manifold would need to be embedded before DFT calculations, losing the speed advantage of a truly two-dimensional model. Instead, DFT calculations are made on a representation of the surface manifold that looks much more like Figure 2.1. A unit length is chosen for all edges, such that all faces form regular polygons (no matter which representation). This yields a truly coordinate-free description, where distances between vertices can be measured by the minimum amount of edges that need to be traversed to go from one vertex to the other.

2.2. Mathematical Representation of Fullerene Graphs

With graphs consisting of vertices and edges, the intrinsic graph information can be conveniently represented in an adjacency matrix A, a matrix of dimensions $N \times N$, if N is the total number of vertices. The entry $(i, j); i, j \in 0, ..., N-1$ in A is 1 if there exists an edge spanning from i to j, otherwise it is zero.

Fullerenes can have anywhere between 20 and hundreds of carbon molecules. This translates to equally as many vertices in the graph. Yet, each vertex has a connectivity of three, so three out of potentially hundreds of entries in a single row of the matrix A will be 1. That is why for a Fullerene graph, an adjacency matrix will always be an ineffective way to save the information, simply because every node only has a degree of three in the cubic representation or a maximum degree of six in the dual representation. In cases like these, it is more efficient to define a *sparse adjacency matrix*. In the case of the dual representation, this matrix has the dimensions $(N \times 6)$. Each line entry represents a vertex and holds the indices of the six connected vertices. In the case of a pentagon node, which only has five connections, the last entry is -1 and can be filtered out whenever accessing the data.

The difference between *adjacency matrix* and *sparse adjacency matrix* are easiest explained with the help of a little example. Figure 2.3 shows a small triangular grid with nine nodes labelled 0 to 8, where 5 is a pentagon node. The adjacency matrix A and sparse

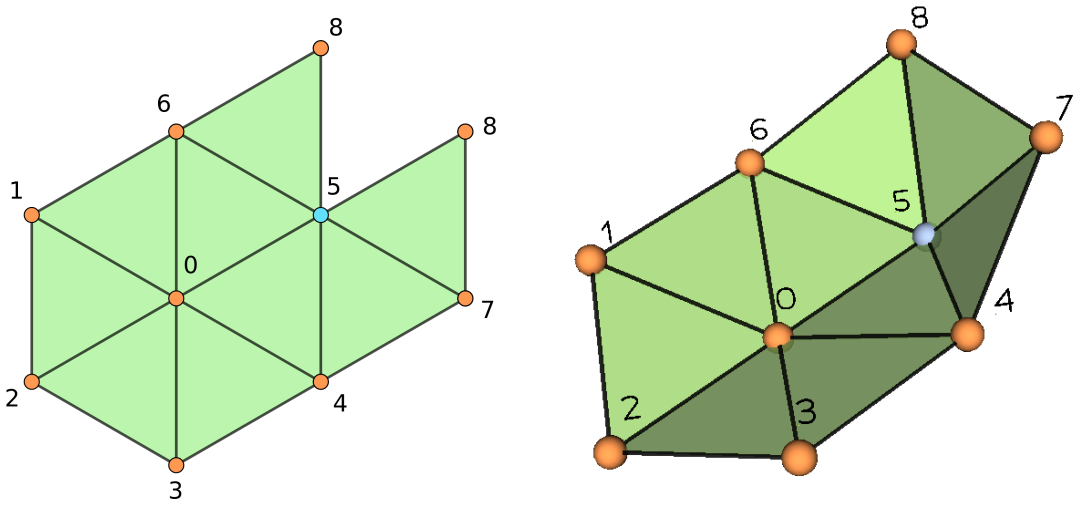


Figure 2.3.: Example for a small hexagonal grid with indices, on the left side flattened in the two-dimensional plane and on the right side embedded in three dimensions.

adjacency matrix S_A of this grid are relatively easy to define.

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}; S_A = \begin{pmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 2 & 0 & 6 & .. & .. & .. \\ 3 & 0 & 1 & .. & .. & .. \\ 4 & 0 & 2 & .. & .. & .. \\ 7 & 5 & 0 & 3 & .. & .. \\ 0 & 4 & 7 & 8 & 6 & -1 \\ 1 & 0 & 5 & 8 & .. & .. \\ 8 & 5 & 4 & .. & .. & .. \\ 6 & 5 & 7 & .. & .. & .. \end{pmatrix}. \quad (2.1)$$

The dots in the sparse adjacency matrix serve as place holders for further connections that go beyond this example. Note that in line five of S_A (counted from zero), there is a -1 entry in the end, as stated before this is because the node only has five neighbours and the negative one serves as a fill-up that can be filtered out by functions that use S_A for calculations.

The sparse adjacency matrix has the ability to hold more information than the normal adjacency matrix. The reason is that the row indices are not bound to be the vertex indices. Instead, one can order the row entries in each line however it is most convenient. For a Fullerene graph, the connected vertices are usually ordered in counter-clockwise order, when looking from the outside of the molecule. Sticking to that ordering is crucial for some functionalities that are needed for the algorithm, as we will see later.

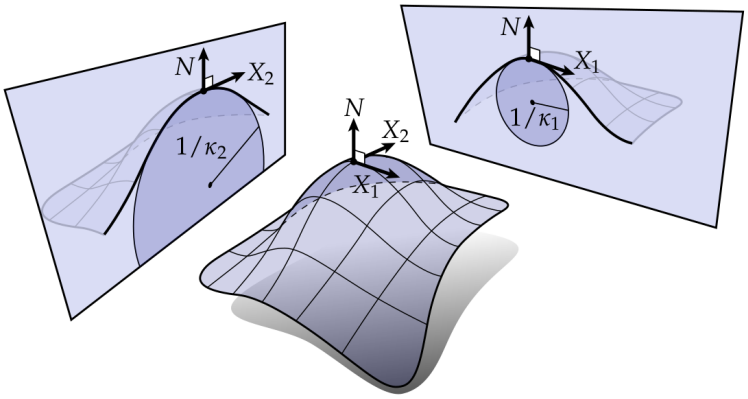


Figure 2.4.: Visialization of principal curvatures κ_1 and κ_2 on an arbitrary surface. Figure from [5].

Going back to the previous example in equation (2.1), take a look at the ordering of lines in the sparse adjacency matrix. Because both sketches in Figure 2.3 are shown from a perspective that would qualify as the *inside* of a Fullerene molecule, the vertices in S_A are ordered in *clockwise* order compared to the Figures.

The sparse adjacency matrix for the dual representation is usually called the matrix of *dual neighbours*. In practice, this matrix is kept at all times, together with the information about the *dual faces*. Technically, this is redundant information, because the faces can be computed from the dual neighbours and vice versa, but both data structures are used regularly and it is beneficial to have the possibility of accessing them at any time without computations.

2.3. Curvature Measures in Fullerene Surface Manifolds

For each point on a differentiable manifold, i.e. a manifold that is locally similar to Euclidean space, one can define principal curvatures κ_1 and κ_2. These are the maximum and minimum curvature of the plane in any direction from the point. This is illustrated in Figure 2.4. Their product, the *Gaussian curvature*, is an intrinsic property of the surface itself

$$K = \kappa_1 \cdot \kappa_2 \tag{2.2}$$

Figure 2.4 shows he principal curvatures of directions $\vec{X_1}$ and $\vec{X_2}$ on an arbitrary surface.
For a more thorough investigation of Gaussian curvature, I refer to Keenan Crane's introduction to differential geometry [5], which figure 2.4 is taken from. It provides a great overview.

Gaussian curvature in a surface point can be positive, zero or negative. In Figure 2.5, three different surfaces are shown. The cylindrical shape in part (a) of the figure does

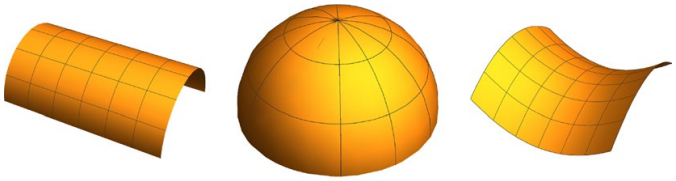


Figure 2.5.: Surfaces with zero (a), positive (b) and negative (c) Gaussian curvature. From [13] with permission.

have extrinsic curvature, but no intrinsic curvature. One of the principal curvatures will be zero, and no point on the surface has Gaussian curvature. The situation in the (b) surface is different, this is a half-sphere and has positive Gaussian curvature. In surface (c), you can see a saddle point that yields negative Gaussian curvature. In Fullerenes, one only encounters nodes that yield a curvature of zero or higher. Negative curvature is not allowed, Fullerenes are convex polyhedra.

The total Gauss curvature of an oriented surface defines its topology through the *Gauss-Bonnet theorem*. Assume a manifold M, a surface element of that manifold $\mathrm{d}A$ and χ the Euler characteristic of the manifold (more about that in a moment). Then the *Gauss-Bonnet theorem* states [5]

$$\int_M K \,\mathrm{d}A = 2\pi \cdot \chi \tag{2.3}$$

This statement has to be dissected a little bit. First of all, the Euler characteristic is a topological invariant of the manifold. Because the manifolds that are treated in this thesis are all Fullerene surfaces, their topology is that of a sphere. Therefore they are closed and orientable, and the Euler characteristic can be computed from the *genus* g of the manifolds. g is always zero for manifolds with spherical topology, so in the context of a Fullerene surface manifold, it can be stated that

$$\chi = 2 - 2g \overset{(g=0)}{=} 2. \tag{2.4}$$

Including this statement, equation (2.3) states that the total Gaussian curvature of a sphere is 4π.

Discrete Fullerene Surface Manifolds

Now, Fullerene surface manifolds are discrete and require a discrete form of the Gauss-Bonnet theorem. The focus is here on simplical surfaces, which restricts the use of the discrete theorem to the dual representation. Nonetheless, some general deductions can be made, which will be addressed later in this chapter.

To be able to define a Gauss-Bonnet theorem on simplical surfaces, one needs a representation for the Gauss curvature K first. This is achieved by defining the *deficit angle* d of a vertex v in the simplical surface

$$d(v) = 2\pi - \sum_{f \in F_v} \angle_f(v). \tag{2.5}$$

F_v is the set of all faces that v is contained in, and $\angle_f(v)$ is the inner angle adjacent to v of a face $f \in F_v$ [5].

Effectively, one is measuring the angle around the vertex with the sum in equation (2.5), and then comparing it to the expected angle, which is 2π. The discrete Gauss-Bonnet theorem relates the deficit angles of all vertices to the Euler characteristic of the surface manifold [5]. Because the number of vertices is finite and they are the only points that can carry curvature, the integral from equation (2.3) reduces to a sum:

$$\sum_{v \in V} d(v) = 2\pi\chi \overset{(g=0)}{=} 4\pi. \tag{2.6}$$

Hereby, V is the set of all vertices in the manifold.

Addressing the manifolds at hand more concretely, one can use the dual representation of Fullerenes as simplical surface. Equation (2.6) fixes the total deficit angle to 4π. Does the dual representation of Fullerenes with a unit edge length condition from chapter 2 obey equation 2.6?

Because of the nature of the dual representation, the vertices can be split up into pentagon vertices v_p with five adjacent faces and hexagon vertices v_h with six adjacent faces. If one also respects the unit edge length condition, all of those faces are equilateral and have inner angles of $\frac{2\pi}{6}$. The deficit angles of these vertices are quite easy to calculate in this model.

$$d(v_p) = 2\pi - 5 \cdot \frac{2\pi}{6} = \frac{\pi}{3} \tag{2.7}$$

$$d(v_h) = 2\pi - 6 \cdot \frac{2\pi}{6} = 0 \tag{2.8}$$

Only the pentagon vertices carry curvature. As addressed in chapter 2, there are exactly 12 pentagons in a Fullerene molecules, equating to 12 pentagon vertices in the dual representation. The total deficit angle adds up to $12 \cdot \frac{\pi}{3} = 4\pi$, which is consistent with the discrete Gauss-Bonnet theorem.

2.4. Alexandrov's uniqueness theorem

A central statement for the shape of Fullerenes is made by *Alexandrov's uniqueness theorem* for convex polyhedra.

Theorem 2.4.1. *Let M be a convex polyhedral metric on the sphere. Then there exists a convex polyhedron $P \in \mathbb{R}^3$ such that the boundary of P is isometric to M. Moreover, P is unique up to rigid motion.*

A convex polyhedral metric is a polyhedral metric that does not have points of negative curvature. This means that both the cubic and dual representation of Fullerenes define a convex polyhedral metric on a sphere. An *isometric* embedding is an embedding that conserves all distances in the surface. Due to Alexandrov's theorem, one can always assume that there is a unique isometric embedding with a rigid form and a total of zero degrees of freedom for any cubic or dual Fullerene surface manifold. However, in this project, only part of the Fullerene surface manifold is constructed, instead of representations of a whole molecule. What value does the theorem have for this work?

In general, any part of a Fullerene surface manifold will have multiple internal degrees of freedom left upon construction. There are exceptions for small areas around pentagon nodes, which will be addressed extensively throughout this thesis, for the first time they will appear in chapter 4. Considering this, one can expect that an embedding of an arbitrary surface manifold region can have anywhere between one and infinitely many representations, depending on the situation. It is to expect that in the case that multiple embeddings are combined to form the whole molecule, there is only one possible way it can be shaped.

One thing that is sure though, is that there always has to be an isometric embedding for any Fullerene surface manifold. If a region of the manifold can not be embedded, it is impossible to find an embedding for the whole manifold.

3. Approximation of Electronic Structure in Fullerenes

From Fullerene bond graph, we compute the structure of the molecule and derive a two-dimensional manifold describing its polyhedral surface formed by hexagons and pentagons. On this manifold, it is possible to compute an approximation of the electronic structure of the molecule. Though the electronic structure of Fullerenes is not a topic that will be addressed further in the process of this thesis, I will give a short introduction because it is important for the motivation behind the work in this thesis.

A Fullerene's electronic structure is a quantum-mechanical multi-body problem, the behaviour of the system is governed by the multi-body Schrödinger equation [13]

$$-\frac{1}{2}\left[\sum_{j=1}^{N}\nabla_j^2-\sum_{a=1}^{M}\sum_{j=1}^{N}\frac{Z_a\cdot e}{r_{i,j}}+\sum_{i<j}^{N}\frac{e^2}{r_{i,j}}-E\right]\Psi=0. \tag{3.1}$$

The problem with the multi-body Schrödinger equation is that it is basically not solvable for molecules bigger than H, $H_2{}^+$ or similar. The equation is a partial differential equation has $3(N_e+N_n)$ degrees of freedom, N_e and N_n being the number of electrons and nuclei. Generally these degrees of freedom are not separable, because the equation features interaction terms between electrons and nuclei. A general expression for equation (3.1) could be written as

$$E\Psi=[T+V]\,\Psi=[T_e+T_n+U_{ee}+U_{nn}+U_{en}+V_{\text{ext}}]\,\Psi. \tag{3.2}$$

Here, T is kinetic energy and V potential energy. V_{ext} represents an external potential, the other three potential energies are the interactions that make the DOF inseparable. As the CARMA project is developing fast methods that have the ability to compute properties of many molecules in a short time, it is necessary to find approximations to this problem that are solvable in little time.

As a first step, one can make use of a widespread approximation in quantum chemistry: the Born-Oppenheimer approximation, declaring the nuclei, which are much heavier and slower than the electrons, to be in fixed positions. This eliminates the kinetic term T_n, the interaction terms U_{nn} are eliminated, while U_{en} terms are now separable and are seen as part of the external potential V_{ext}.

3.1. Density Functional Theory

This section follows the DFT developed in [2]. There is still the problem of the inseparable terms U_{ee} and the number of DOF cannot be reduced under $3N_e$. This can be attacked by modifying the form of the problem further, such that it can be viewed in the framework of a *density field theory* (DFT), based on the two *Hohenberg-Kohn theorems* that were proposed in [9]. Still assuming a quantum-mechanical system of N electrons, with Hamiltonian

$$\mathsf{H} = T + U_{ee} + V_{\text{ext}}, \tag{3.3}$$

the *electron density* ρ_0 of a ground state Ψ is defined as

$$\rho_0 = \rho[\Psi_0](\boldsymbol{x}) = \int \mathrm{d}\boldsymbol{x}_2 ... \int \mathrm{d}\boldsymbol{x}_N \; |\Psi_0(\boldsymbol{x}, \boldsymbol{x}_2, ..., \boldsymbol{x}_N)|^2. \tag{3.4}$$

Equation (3.4) shows, how ρ is determined by the ground state wave function of the system Ψ. The first Hohenberg-Kohn theorem states that the inverse is also true, that is the ground state wave function of a system is *uniquely* determined by the ground state density ρ. That means that one can not only see the density as functional of the wave function, $\rho[\Psi]$, but that it is also possible to define $\Psi = \Psi[\rho]$. This consecutively leads to a functional

$$A = A[\rho] = \langle \Psi[\rho] | A | \Psi[\rho] \rangle \,. \tag{3.5}$$

for an arbitrary operator A. An energy functional can be defined by

$$E[\rho] = \langle \Psi[\rho] | \mathsf{H} | \Psi[\rho] \rangle \,. \tag{3.6}$$

The Hamiltonian is still the one from equation (3.3). From the first two terms of this Hamiltonian, the functional

$$F[\rho] = \langle \Psi[\rho] | T + U_{ee} | \Psi[\rho] \rangle \tag{3.7}$$

can be defined. The last term, the external potential, can be split up into single-electron terms

$$V_{\text{ext}} = \sum_{i=0}^{N} \mathsf{v}_{\text{ext}}(\boldsymbol{x}_i). \tag{3.8}$$

For the functional in equation (3.6), this yields

$$V_{\text{ext}}[\rho] = \langle \Psi[\rho] | V_{\text{ext}} | \Psi[\rho] \rangle = \int \mathrm{d}\boldsymbol{x} \, \rho(\boldsymbol{x}) \mathsf{v}_{\text{ext}}(\boldsymbol{x}). \tag{3.9}$$

Equation (3.6) can then be rewritten as

$$E[\rho] = F[\rho] + \int \mathrm{d}\boldsymbol{x} \, \rho(\boldsymbol{x}) \mathsf{v}_{\text{ext}}(\boldsymbol{x}). \tag{3.10}$$

The second Hohenberg-Kohn theorem states that the energy functional (3.10) reaches its minimum exactly when $\rho = \rho_0$; in other words, the variational principle can be used to find the ground state energy from the functional (3.10) [9]. This means that now there is a clear path to compute the ground state energy from the ground state density. Only, the functional $F[\rho]$ is not a known quantity and finding an approximate form of $F[\rho]$ is an unsolved, and possibly unsolvable problem. One can use the *Kohn-Sham* approach, which proposes that for each Hamiltonian with an electronic interaction term, like in (3.3), and ground state density ρ_0, there exists a non-interacting Hamiltonian $\mathsf{H}_0[\rho] = T + V^{KS}[\rho]$, which has the same ground state density ρ_0. This breaks down the many-electron Schroedinger equation into a set of single-electron equations, coupled through the Kohn-Sham potential.

$$\left[-\frac{1}{2}\nabla^2 + \mathsf{v}^{KS}[\rho](\boldsymbol{x})\right]\phi_i(\boldsymbol{x}) = \epsilon_i\phi_i(\boldsymbol{x}) \text{ with } \rho(\boldsymbol{x}) = \sum_{i=1}^{N}|\phi_i(\boldsymbol{x})| \tag{3.11}$$

This yields the approximate kinetic energy $T^{KS}[\rho]$. Taking a mean-field approach for the electron-electron interaction U_{ee}, one can sum up the inaccuracies that arise from the approximations in the *Exchange-Correlation energy*

$$E^{XC}[\rho] = F[\rho] - T^{KS}[\rho] - U_{ee}^{\mathrm{mf}[\rho]}. \tag{3.12}$$

With some more calculation, one reaches the point where the Kohn-Sham potential is

$$\mathsf{v}^{KS}(\boldsymbol{x}) = \mathsf{v}_{\mathrm{ext}} + \mathsf{v}_H(\boldsymbol{x}) + \mathsf{v}_{XC}(\boldsymbol{x}), \tag{3.13}$$

where the external potential is assumed known. v_H is the *Hartree potential* from the mean field approximation and v_{XC} is the exchange-correlation potential, which is not known.

The sense of the exchange-correlation energy is that it is much smaller compared to $F[\rho]$ than U_{ee} or T^{KS}. Nevertheless, it still needs to be approximated in order to find a good representation of $F[\rho]$. This would go beyond the purpose of this chapter though, so for now we finish the topic of DFT by writing the *Kohn-Sham equations*, combining (3.11) and (3.13):

$$\left[-\frac{1}{2}\nabla^2 + \mathsf{v}_{\mathrm{ext}} + \mathsf{v}_H(\boldsymbol{x}) + \mathsf{v}_{XC}(\boldsymbol{x})\right]\phi_i(\boldsymbol{x}) = \epsilon_i\phi_i(\boldsymbol{x}) \tag{3.14}$$

The function of a two-dimensional surface DFT is not only to be a fast approximation. There exist fast graph-based methods for electronic structure that achieve high computation speeds, for example the tight-binding model, Hückel's method for π-orbitals in molecules or also chemical graph theory approaches. However, these methods do not reach the accuracy that a DFT can reach trading it off for a higher computation speed. The goal of a two-dimensional DFT is to combine high computation speeds of microseconds per atom and a linear size scaling with the higher accuracy that a normal three-dimensional DFT can provide. Equipped with high computation speed and

a good accuracy, the goal is to predict molecule properties of many molecules in the enormous isomer space of Fullerenes in a short time. The coming section will introduce a finite element method to represent the equations from this DFT in two dimensions and coordinate-free, such that the speed-benefit can be ensured.

3.2. The Finite Element Method

Some big steps have been taken to reach equations (3.14), but right now, all quantities still depend on a global coordinate system. This would require an embedding of the full molecule, which is exactly what should be avoided in this context. To reach a description on the two dimensional manifold, a *Finite Element Method* (FEM) can be developed for the Fullerene surface manifolds [1].

Finite element methods are based on dividing some *domain* Ω into multiple *cells* c, defined as disjoint subdomains of Ω such that $\bigcup_1^{N_{\text{cells}}} c_i = \Omega$. It is very convenient that when choosing a Fullerene surface manifold as domain, the dual representation provides a set of triangular cells that can be used for the purpose of FEM without having to define a mesh by hand.

On each of these cells, functions are approximated by a set of polynomial basis functions $\mathfrak{e}_i(\boldsymbol{x})$. Hereby, $\boldsymbol{x}$ does not correspond to the global three-dimensional coordinate system, but to a two-dimensional coordinate system (x, y) chosen for the cell alone. The discretization of this system is done by restricting each basis function to a single *node* z_j on the cell, such that it effectively works as Kronecker delta $\mathfrak{e}_i(z_j) = \delta_{ij}$. An arbitrary function f can then be expressed by $f = \sum_i f_i \mathfrak{e}_i$ with coefficients $f_i = f(z_i)$. The *nodes* z_i are by the way not generally the same as the nodes or vertices of the surface manifold. It depends on the order of the polynomial basis functions, how many *nodes* are needed. There will be one basis function for each node. For the remainder of this chapter, I will use *nodes* when referencing the reference points for the polynomial basis functions.

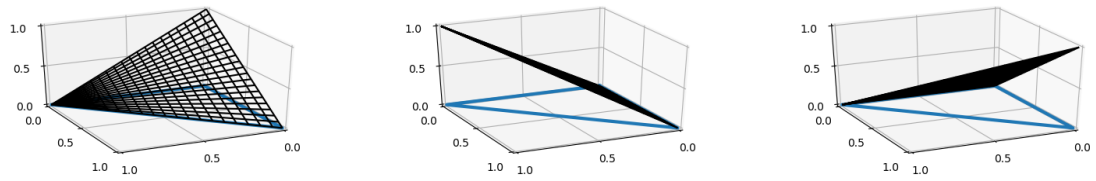


Figure 3.1.: Linear shape functions of a triangular cell. There are three functions, as the cell is defined by three corner nodes. From [14] with permission.

Defining three or more basis functions for each cell sums up to quite a big amount of work, if done separately for every cell. To avoid this, a *reference cell* is defined, representative for any arbitrary cell in the mesh. On this reference cell, one can define all the nodes needed for the polynomials of a given order, as well as a set of *shape functions* for the

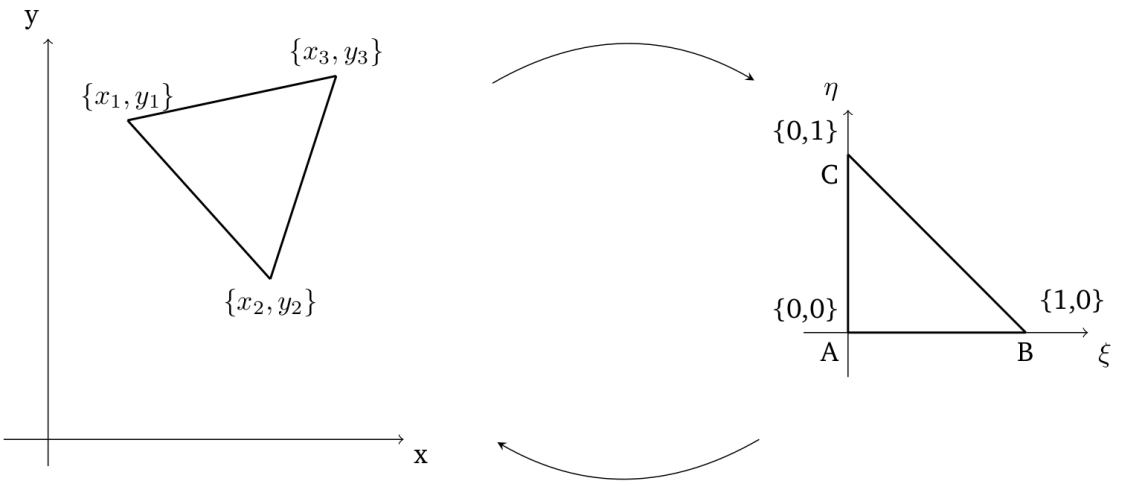


Figure 3.2.: Triangular reference cell with coordinates (ξ, η). Each individual cell can be represented by a coordinate transformation to local cell coordinates (x, y). From [14] with permission.

nodes, which are the basis functions of the reference cell. Let (ξ, η) be the coordinate system of the reference cell, then the nodes and basis functions of any cell in the mesh with coordinate system (x, y) are defined by doing a coordinate transform $(\xi, \eta) \to (x, y)$ on the reference cell and its shape functions.

Coming back to tailoring the *Kohn-Sham equations* to the Fullerene surface manifold, one can take another look at equations (3.14). I have introduced how the wave functions can be represented in terms of cell basis functions, but what about the rest of the equation? To transform (3.14) into an equation that is completely coordinate free, one typically brings it to the *variational formulation* first. This essentially means to introduce an arbitrary test function θ by multiplying it to the equation and partially integrating the kinetic term, such that equation (3.14) becomes

$$-\frac{1}{2}\int_\Omega \mathsf{d}^n x\ \nabla\theta(\boldsymbol{x})\nabla\phi(\boldsymbol{x}) + \int_\Omega \mathsf{d}^n x\ v_{\text{eff}}(\boldsymbol{x})\,\theta(\boldsymbol{x})\phi(\boldsymbol{x}) = \epsilon\int_\Omega \mathsf{d}^n x\ \theta(\boldsymbol{x})\phi(\boldsymbol{x}). \tag{3.15}$$

Hereby, I have summed up all potential in the effective potential v_{eff}. θ and ϕ can then be expanded in terms of the mesh basis functions

$$\theta(\boldsymbol{x}) = \sum_l \theta(z_l)\mathfrak{e}_l(\boldsymbol{x}) \quad \text{and} \quad \phi(\boldsymbol{x}) = \sum_k \phi(z_k)\mathfrak{e}_k(\boldsymbol{x}). \tag{3.16}$$

Inserting the expansions of θ and ϕ into equation (3.15) and cancelling the terms appearing from θ on both sides of the equation yields

$$\sum_k \phi(z_k)\left[-\frac{1}{2}\int_\Omega \mathsf{d}^n x\ \nabla\mathfrak{e}_l\nabla\mathfrak{e}_k + \int_\Omega \mathsf{d}^n x\ v_{\text{eff}}\,\mathfrak{e}_l\mathfrak{e}_k\right] = \epsilon\sum_k \phi(z_k)\left[\int_\Omega \mathsf{d}^n x\ \mathfrak{e}_l\mathfrak{e}_k\right]. \tag{3.17}$$

This can be written as a matrix equation

$$[\boldsymbol{L} + \boldsymbol{V}_{\text{eff}}]\,\boldsymbol{\phi} = \epsilon \boldsymbol{M}\boldsymbol{\phi}, \tag{3.18}$$

where the matrix elements $\boldsymbol{L}_{kl}$, $(\boldsymbol{V}_{\text{eff}})_{kl}$ and $\boldsymbol{M}_{kl}$ represent the integrals from equation (3.17) and $\boldsymbol{\phi}$ is a vector consisting of elements $\boldsymbol{\phi}_k = \phi(z_k)$. These integrals can be evaluated individually over each cell and therefore don't depend on the global coordinates. Also, the equation to deal with is now a matrix equation and therefore easily representable numerically.

In summary, the introduction of this finite element method reduces the previously developed DFT, which is a high accuracy approximation for the electronic structure of Fullerenes, from describing the molecule in a three-dimensional state to describing the electronic structure purely on the surface of the molecule. This yields significant speed benefits, as the new theory is only two-dimensional and also coordinate-free. However it comes with costs in the accuracy due to significant out-of-surface interactions, for example in the high-curvature areas of the molecule. That leads us back to this thesis project: To capture the most significant out-of-surface interactions, a mixed-dimensional approach is suggested. High-curvature regions in Fullerenes are constructed with a high-speed method in approximate shape, to increase the accuracy of the surface DFT. The first challenge is to develop a construction method for high-curvature "pocket regions" in Fullerene molecules, starting from the graph bonds that are used for two-dimensional coordinate-free description

Constructing a Local Embedding Algorithm

4. Structure of Embeddings

This chapter marks the beginning of the method development that has been done in the course of this thesis. Chapter 1 has shown that a construction process for single pocket region embeddings of surface manifolds has to fulfill a set of different requirements. This chapter will focus on introducing basic mechanisms for describing the construction process, such it respects these requirements. The main focus will first be to find a way of describing the small, very local embeddings that are created initially. The sections 4.1 and 4.2 address this in dual and cubic representation. Afterwards, the transfer to bigger structures that can describe pocket regions of a molecule will be planned in section 4.3. This chapter focuses on the structure of the single embeddings, how the construction can be linked is the focus of chapter 5, which completes the method development. From chapter 6, the implementation will be the focus.

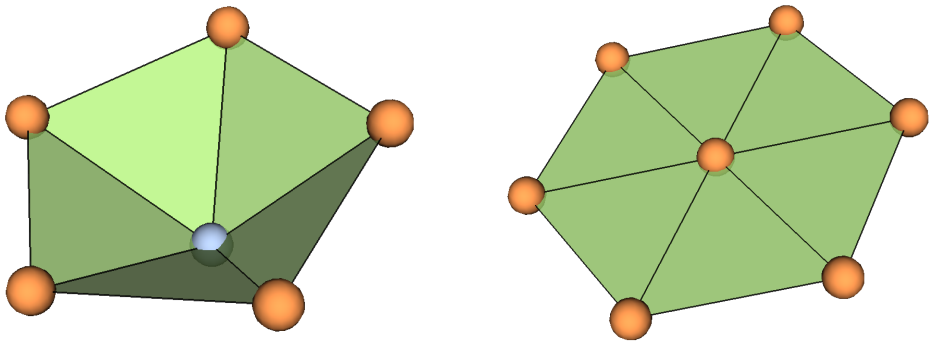
4.1. Dual Structure

In a dual surface manifold embedding with equilateral triangles, pentagon nodes cause a cone structure that will be very useful in construction. To compare embedded regions around hexagon nodes and around pentagon nodes, embeddings of a pentagon node and a hexagon node with their adjacent faces and neighbour nodes are shown in Figure 4.1. Hexagon nodes do not infer very specific shape constraints. Even when all triangles can only be equilateral, there are three edges along which the embedding in Figure 4.1b can be bent, without actually introducing any curvature to the center vertex. These three axes are the three pairs of edges crossing the center node in a straight line. This embedding can be shaped in many ways.

Contrary to that, every pentagon node creates an embedding with cone shape. The shape in Figure 4.1a is the same for every pentagon node. This means that, while a region of the surface manifold that consists of hexagon vertices can in general be shaped in different ways, a region that contains a pentagon vertex, *always* contains that specific shape.

Because of the importance of these small regions around pentagon nodes, they are named.

Definition 4.1.1 (Minimal Size Patch)**.** An embedding of a pentagon node, its five neighbour nodes and the corresponding faces and edges is called a *minimal size patch*.



(a) Embedding of pentagon node (blue) and neighbours.

(b) Embedding of a central hexagon node surrounded by neighbours.

Figure 4.1.: Comparison of hexagon and pentagon nodes in embeddings: 4.1a is a rigid object, while 4.1b can still be re-shaped.

More generally, embeddings that evolve around a single pentagon node, are called *patches*. A specific definition for that will follow later.

Constructing Embeddings from Minimal Size Patches

Pentagon nodes do not only generate minimal size patches around them, they also have great influence on the shape of the embedding beyond the minimal size patch. An embedding with a pentagon in the center keeps the cone shape. A good way to build up bigger embeddings is to do a layered construction around a minimal size patch. This concept is extremely important for the construction method because it is not only applied for growing a patch, but also for the cone structure that the whole pocket region will have later, which also has to be grown in layers from a center. More about that in later chapters. Figure 4.2 shows the construction of two more layers of nodes, marked by the thick red and blue lines, around a minimal size patch that is marked in green. With each layer of nodes comes a set of new faces, marked in the same colors. A *layer* of nodes can be defined as follows:

Definition 4.1.2 (Node Layer)**.** A layer of nodes is a set of nodes, which can be reached by traversing a single edge from any node of the previous layer, while not being in any existing layer of nodes in the same embedding.

That means that some set of nodes needs to be defined as the initial layer, which is equal to the *center* of the object that is being grown. For patches like in Figure 4.2, the center is always the pentagon node. Therefore the first layer is the boundary of the minimal size patch (thick green line), and from there on, one can count the layers to obtain the *radius* of the patch. For example, the patch in Figure 4.2 has a radius of $r = 3$.

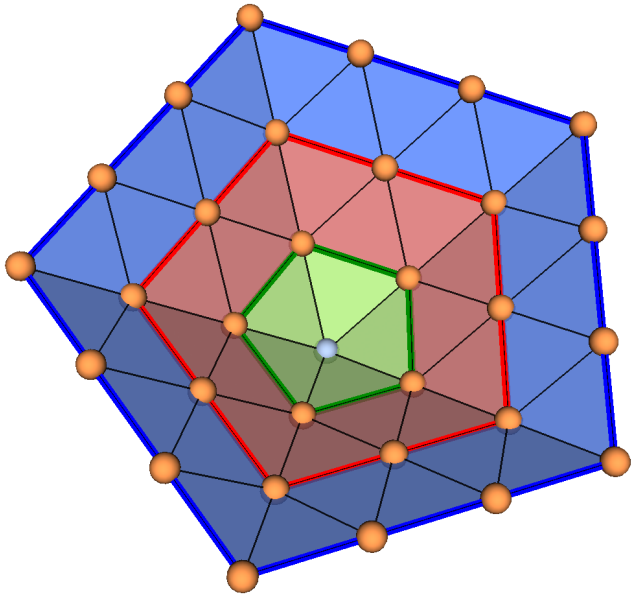


Figure 4.2.: Minimal size patch (green) with two more layers of nodes (marked by red and blue lines).

This layered construction, which gives the patch a radius, is highly important for the structure of a patch. Therefore, the term *patch* is defined as follows:

Definition 4.1.3 (Patch)**.** A three-dimensional embedding of a region in a Fullerene's surface manifold in dual representation is called a *patch*, if it fulfills the following properties: A patch has to be centered around a pentagon node and be built in layers, such that a *radius* can unambiguously be defined for the patch by counting the number of edges traversed from the center pentagon to any node in the boundary of the patch. Except for the center pentagon node, a patch can not contain any further pentagon nodes.

Sectors and Corners

The description of patches requires some more structure than layers. This is especially important for the technical description of pocket regions and their cone shape in the late stages of the algorithm (this is addressed in detail in the chapters further down this chapter). Therefore, it is useful to introduce *corners* and *sectors*.

On the example patch from Figure 4.2, it is quite intuitive to introduce the concept of corners and sectors. In a patch like that, the corners, which are marked in red in Figure 4.3, run along the actual physical corners of the cone-shape. The sectors are the areas between the corners. The nodes in an embedding can be split into two groups: Nodes that are on a corner and nodes that are in a sector, that is *sector nodes* and *corner nodes*.

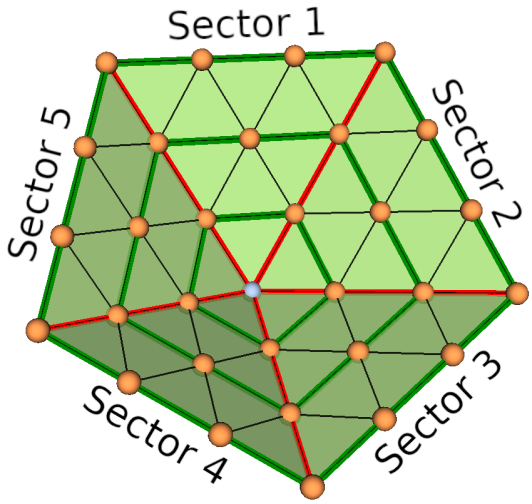


Figure 4.3.: A patch with radius $r = 3$, the three layers are marked in green lines. The corners of the patch are marked in red and divide the faces into five sectors.

While sectors and corners can be told intuitively in the previous example, there will be situations where a more mathematical definition is necessary. At this point, the layer structure that was defined above comes in handy: In Figure 4.3, all corner nodes in a layer are only connected to the previous layer by a single edge. This edge will always be one of the edges marked in red, which build the corners. Contrary to that, all sector nodes are connected to the previous layer by two edges. Therefore corner nodes and sector nodes can be defined as follows.

Definition 4.1.4 (Corner Node)**.** A corner node in an embedding is a node that is only connected to the previous layer by a single edge. A corner is therefore made of a set of corner nodes, each in their own layer, and their connecting edges.

Definition 4.1.5 (Sector Node)**.** A sector node is a node that is connected to the previous layer by two edges. Following that, a sector is a collection of neighbouring sector nodes, bounded by corners.

The patch structure of layers, nodes and sectors being in place, makes it possible to revisit the process of adding layers to a minimal size patch. A detail that I quietly ignored is how to obtain the coordinates of the new nodes. These nodes are in general not fully determined, so some decisions need to be taken. The patches in Figures 4.2 and 4.3 are built after the simple principle of continuing corners straight and keeping the sectors flat. This is what is called a *standard shaped patch*. A detailed method for the construction of standard shaped patches is addressed in chapter 8.4.

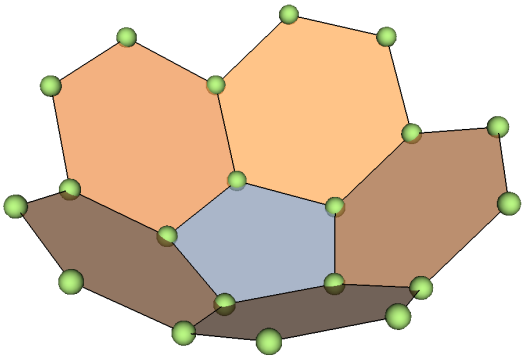


Figure 4.4.: A minimal size patch in cubic representation.

4.2. Cubic Embedding Structure

Compared to the dual embeddings, it is harder to obtain a nice structure for cubic embeddings. The advantage of the cubic representation is of course that it is the physical form of the molecule. Therefore it can never be fully forgotten, because the end result of a dual embedding needs to be representable in the cubic form as well. Considering how easy it is to change between cubic and dual embeddings though, obtaining the result in the dual representation and then transferring it to the cubic representation does not pose a huge challenge.

Nevertheless, the following paragraphs will show some examples for embeddings in the cubic representation and why it is harder to work with. The beginning can be made by the *minimal size patch* of the cubic embedding. An example is shown in Figure 4.4, where all faces are regular polygons and therefore all edges have the same length. So far, no problems arise with the construction of the cubic representation embedding. This changes when adding another layer to the cubic minimal size patch though. It is not possible to construct a second layer without either bending some faces or giving up the regular polygon shape. Figure 4.5 shows both situations. In the left embedding, five of the ten faces in the second layer have been bent to continue the corners of the cone. In the right embedding, the second layer is not fully constructed, instead only the five faces that were bent in the left embedding, are constructed flatly instead. This yields non-regular hexagons, and theoretically a second set of differently shaped irregular hexagons to complete the second layer. Both options would already make a three-dimensional embedding more difficult than it is in the dual representation. Additionally, it is not as easy to construct a nice structure for the patches. Layers can of course be constructed, but the connections between nodes in a layer don't always correspond to edges in the mesh, like they do in the dual representation. Sectors and corners can only be defined in the model with bent faces, although only based on the cone shape and not based on the amount of connections to the previous layer. In summary, while the cubic representation is the more physical representation, its practicality lacks in a model with a global rule

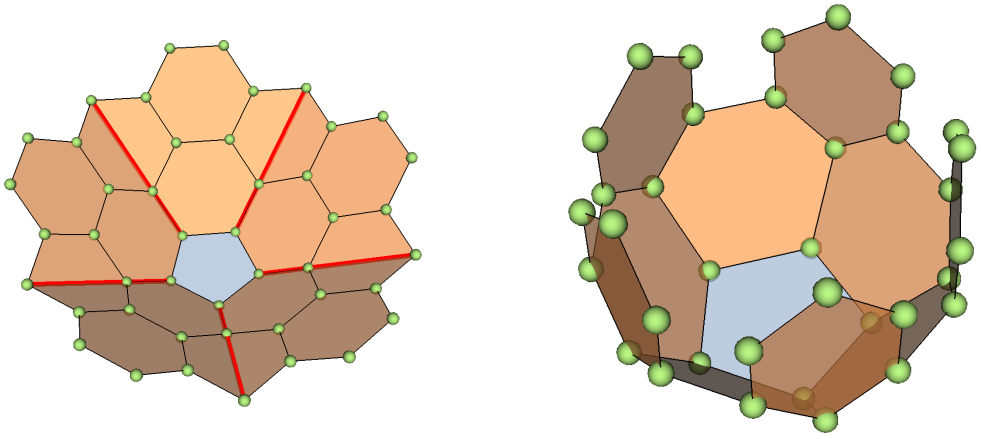


Figure 4.5.: Patch of cubic faces embedded around a center pentagon face in two different approaches. Once, bent faces were allowed (left) and once they weren't (right).

for edge lengths and therefore working with the dual representation is advantageous.

4.3. Combining Patches - The Pocket Region

From the cubic representation, we go back to the dual representation embeddings. Earlier in section 4.1, the structure of patches was built up. This structure needs to be transferred to embeddings that can hold more than a single pentagon node. In the introductory chapter, such structures were referred to as "pockets" or "pocket regions", and this name will now get a formal definition.

Definition 4.3.1 (Pocket Region)**.** A three-dimensional embedding of a region in a Fullerene's surface manifold in dual representation is called a *pocket region*, if in consists of 1-6 pentagon nodes and an arbitrary number of hexagon nodes, carrying Gauss curvature of less or equal to 2π.

A pocket region can contain at minimum one patch, which is the minimum to justify an embedding, and up to a maximum of six patches, which is the case for the ends of nanotube-like Fullerenes like in Figure 1.3. This already implies that the patches can not be selected arbitrarily; grouping patches to build a pocket region from them has to fitted to the actual distribution of pentagon nodes in the molecule. An easy example for selection of pocket regions is shown in Figure 4.6. The $C_{120}-T$ in this figure has a very regular structure and there are four clear groups of three pentagon nodes each. In the Figure, each of those groups is bordered in red; in the shape that is built by the minimum size patches of the three pentagon nodes. This is the only valid way to group patches

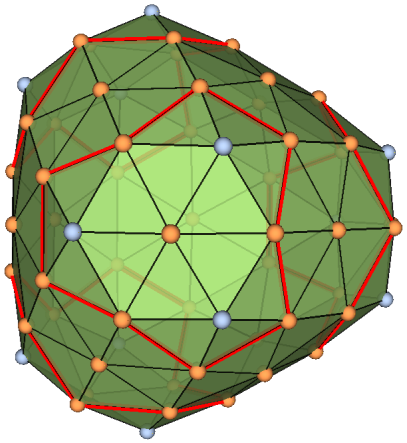


Figure 4.6.: $C_{120}-T$ molecule in dual representation. It has four groups of three pentagon nodes, where the coombined minimum size patches of these groups have been circled in red.

to pocket regions in this molecule. In other molecules, the selection process might not always be as clear, but generally there is an optimal solution.

Now, how does one build and structure pocket regions? The pocket region build can not be started by building a single patch and then including additional pentagon nodes on the way. Otherwise, the shape influence of those nodes on the molecule is not given in all radial directions around the nodes. Similarly, construction can not be started by choosing a center node for the pocket region and doing a layered construction around it. Instead, the patches need to be constructed first up to a radius where they overlap with at least one full face, and then they can be combined to form the raw shape of a pocket region. In the case of the $C_{120}-T$ molecule from above, to reach an overlap between the three patches, one patch has to be constructed with a radius of $r = 2$, its boundary is shown in red in Figure 4.7, and the other two with a radius of $r = 1$ (blue and yellow boundaries). It is not necessary that the blue and yellow patch also overlap, as both can just be combined with the red patch.

Furthermore, pocket regions can essentially carry all the structure that patches also carry - that is cones, edges and layers. For this, the only precondition is that a fitting center has been selected. Contrary to patches, this can not always be a single node. Returning to the $C_{120}-T$ molecule one last time, the center of every pocket region in this group would be the hexagon node in the middle of three pentagon nodes. Starting from this center node, one can compute the layers and find the corners, as it is shown in Figure 4.8. The layered structure also gives the pocket region a radius, and somewhat of a cone structure. This structure was already mentioned in the introduction chapter as a requirement, and therefore it is strictly required to check this structure in all pocket regions. Corners and layers will be discussed in greater detail in chapter 10.

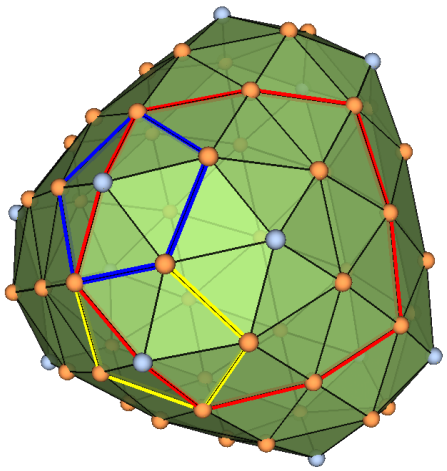


Figure 4.7.: $C_{120}-T$ molecule in dual representation. The necessary patch radii to combine three patches in a group to a pocket region, are $r = 2$ (marked in red) and twice $r = 1$ (blue and yellow).

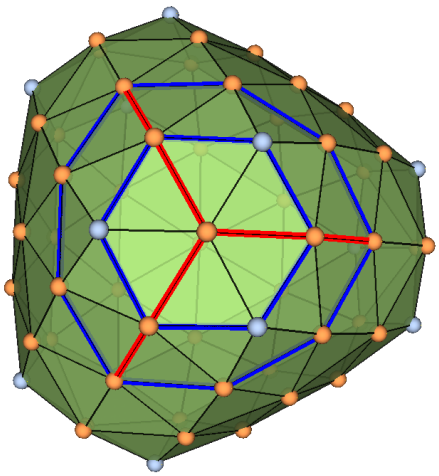


Figure 4.8.: $C_{120}-T$ molecule in dual representation. The corners (red) and first two layers(blue) of one of the four identical pocket regions in this molecule.

5. Finding a Promising Approach

After a lot of introduction and many theoretical considerations, it is necessary to stop for a moment and summarize the important points, such that they can be used to draw a concept sketch of what this thesis is trying to construct.

In chapter 3 the basics of an electronic structure theory are laid out, designed to work in two-dimensional Fullerene surface manifolds. Developing such a theory is desirable, because it is computationally fast and can be applied to many Fullerene isomers in short time. However, the two-dimensionality that makes a theory like that fast, also comes with a caveat: The precision of the approximation decreases because of the reduced dimension. This becomes relevant when ignoring out-of-surface interactions that appear already in relatively short distances, such that they can not be neglected due to the $1/r^2$ scaling of the Coulomb force. Excluding special cases with very flat or nanotube-like molecules, out-of-surface interactions have the hightest relevance in the high curvature regions of a surface manifold. The distance measured for an interaction in these regions can vastly differ from the actual three-dimensional distance between two interacting particles, rendering the strength of the interaction smaller than it should be.

An approach to improve the accuracy of the approximation can be to - partly - return to a 3d-ified representation of the molecule by embedding parts of the surface manifold in three dimensions. While a 3d-ified representation of the surface manifold to approximate interactions more accurately sounds great, one might ask if that does not exactly nullify the computation speed gained by reducing the dimensionality of the problem in the first place. To avoid building a method that might turn out too computationally heavy at a later stage, I want to stick to two main concepts:

1. The embedding is built only under use of geometrical constraints, using the bond graph given from the coordinate-free surface manifold.

2. Single regions of the surface manifold can be constructed, without having to construct the remainder of the molecule.

The use of concept number one potentially increases the speed compared to for example *Force Field Optimization* which uses, as the name says, approximations of the forces acting on nuclei and electrons in the molecule to compute the optimal length of every edge and determines the molecule structure this way. Concept number two is strictly necessary to ensure that the the computation speed for bigger molecule sizes continues to scale with the two-dimensional surface manifold, instead of in a three-dimensional way.

5.1. General Construction Requirements

Over the course of the first few chapters, many details of the description of Fullerenes have been discussed, containing useful information for the draft of a construction method for pocket regions. This section aims to summarize this information and provide some structure for discussing the decisions that have to be taken.

The introduction chapter is arguably the most important source for construction concepts and requirements. The concepts drawn for this chapter are fundamental to the design of the method Just before the beginning of this section, two general concepts were already noted with respect to computation speed: that the method is based on the bond graph and that it can embed single regions independently. A more physical requirement is the even shape of the end product, which should resemble a cone-like structure with an even radius, to ensure that the cutoff for the interaction strength of out-of-surface interactions can be even. This immediately sets a requirement for the late stage of this thesis: An approach to bring embeddings to an even shape, and to obtain embeddings in any desired size is needed, such that the cutoff can be chosen freely due to criteria that are independent from the method itself.

Before getting to that, the construction needs to be started though. The easiest approach would be to simply choose a point on the surface manifold that has a very central position w.r.t the pentagons in the region. This point can serve as the center of the cone shape for the late stage embedding from above, it might as well serve as starting point for the construction and one can construct the whole cone layer by layer. However, this approach neglects important shape features stemming from pentagons coming along during the construction. Chapter 4 demonstrated the shape influence of pentagons in a bond graph based construction. This influence needs to be respected for a good approximation of the physical molecule shape.

The assumption that pentagons or pentagon nodes shape the manifold, but specifically their immediate surroundings, does not only require to start constructing from one or a few pentagon nodes. A surface region has to be constructed around every single pentagon node that is supposed to be in an embedding, and only as a second step, bigger regions can be built as combinations of smaller ones.

In summary, these requirements guide the overall structure of the method to consist of three main steps: First, embeddings need to be constructed around pentagons, then combined to form a pocket region, and at last the pocket region has to be brought to an even cone shape and there has to be the opportunity to grow it further, beyond the size provided by the combination process.

Another requirement for construction is to choose suitable edge lengths for the embedding. Embedding the coordinate-free two-dimensional surface manifold into three-dimensional space requires a clear specification of edge lengths. In other construction methods, the edge lengths in a surface manifold might be calculated due to certain protocols, for example *Force Field Optimization* optimizes edge lengths to minimize forces inside the

molecule. This requires to take into account physical forces, in order to decide on the lengths of individual edges, an option that doesn't exist for the method that is being built in this thesis.

Instead, a global rule for all edges in a manifold has to be drafted. There are various options. So far, only the easiest one has been touched on, which is to have only regular polygons as faces, yielding unit length edges. Choosing an appropriate set of edge length constraints should not be done without being clear about which representation is chosen for the embedding. Requiring regular polygon faces in the cubic grid does not mean the same as in the dual grid, therefore the next sections will compare, how well dual and cubic representation can be used for three-dimensional embedding and which edge lengths to choose.

A Short Note about Edge Lengths

So far in this thesis, all examples of embeddings were drafted assuming equilateral triangles in the dual representation and regular polygons in the cubic representation, both of which have as a result that all edges have the same length. This is both the easiest method and, in the case of the cubic representation, also the straightforward continuation of the coordinate-free, two-dimensional Fullerene surface manifold used in the CARMA project, which assumes equilateral triangles. There exist other possible rules for the edge lengths though. In the dual representation, edges connect hexagon and pentagon nodes, which gives the three combination options hxg-hxg, hxg-ptg and ptg-ptg. In the cubic representation, it is the same, only that a hxg-hxg edge is shared by two faces, instead of connecting nodes of some type.

Considering that chapter 4 showed that the dual representation works very well with equilateral triangles and there is no reason to change the model. The requirement of equilateral triangles can be dropped at any time if needed, so it is always better to go with the condition first.

For the sake of completeness I want to mention that in the cubic representation, choosing other edge length conditions does not seem to change the practicality of embeddings in any way. Therefore, after some initial approaches to make a cubic representation embedding work, the focus of this thesis project was put exclusively on dual representation surface manifolds.

5.2. Towards an Algorithm Concept

Building up a surface embedding for Fullerene surface manifolds takes many small steps and it is quite easy to get lost in the method that is proposed in this thesis. This chapter serves as a description of the algorithm concept, that is the structure of the algorithm based on the theoretical foundation from recent chapters.

The comparison of representations and edge length conditions has a pretty clear outcome. It is most promising to develop a model for embedding surface manifold regions in three-dimensional space based on

1. The dual representation of the surface manifold and

2. Equilateral triangles as faces.

For process itself, the coarse structure is the result of the considerations made in chapter 5.1, consisting of three main steps.

The Construction Step

First, the embeddings around single patches need to be constructed from the bond graphs. Because one needs to know, which pentagon nodes belong to which pocket region, this step will at first group all the pentagon nodes into pocket region groups. If pentagon nodes are grouped together, that means that their patch embeddings are supposed to be combined to a pocket region embedding. Both of these concepts will be addressed in great detail in chapter 8. After the radii for all twelve patches are known, the key piece of this construction step begins: The patches are constructed. Figure 5.1 show a visual overview of what happens in the construction step. Note that this procedure happens

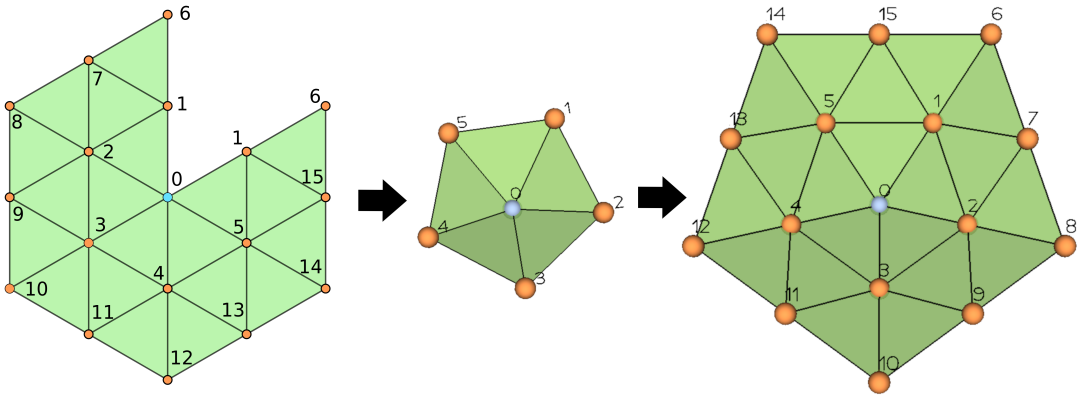


Figure 5.1.: Visualization of the Construction Step: From the bond graph, first minimal size patches are constructed and then grown to the radius that is needed for combination.

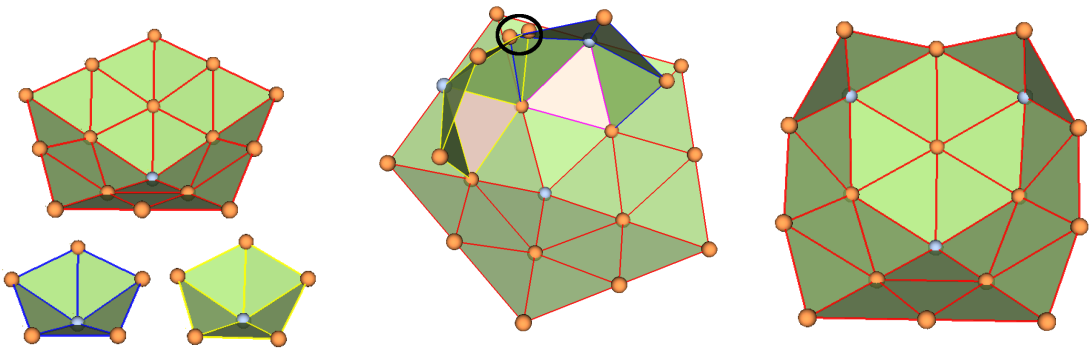
for all twelve patches in a molecule. What is not shown are the more abstract operations of grouping patches and determining their radii, which happen before the embeddings are generated. These are also important parts of the construction step.

The Combination Step

Next, the single patches have to be combined to form pocket regions. This is quite an intricate process as well. In a first step, one of the existing patches is selected to begin the pocket region embedding. Then all patches that have an overlap with this pocket region can be merged into the embedding. This process is continued until all patches are merged, which can be a maximum of six patches in total.

The first step of the merging process is to place a patch by simply selecting one of the overlapping triangles between patch and pocket region and then use translation and rotation operations on the patch coordinates, such that the positions of the three nodes defining the overlap triangle coincide. Sometimes, this generates a valid geometry directly, meaning that upon adding the new nodes and edges to the pocket region, all geometrical constraints are still obeyed. In other cases, an edge length or some node positions are not correct and the geometry of the pocket region is invalid. Then a *geometry fixing* has to be executed, which basically adjusts the position of the patch, such that the geometry becomes valid again. The details on this are addressed in section 9.2.

In the third step of the merging process, the new triangles and edges are added to the pocket region and the patch is added to the pocket region. In the end of this step, the raw pocket regions with all shape-defining elements are done, but they generally are not in a cone shape with intact layers yet. Returning to the example of C_{120}–T, which has already been used for the pocket region structure in chapter 4.3, Figure 5.2 shows the main steps. The input from the construction step is one patch with $r = 2$ (red) and two



(a) (b) (c)

Figure 5.2.: Visualization of the construction step on the example of a C_{120}–T pocket region featuring three patches. The central patch is the $r = 2$ patch in (a), blue and yellow are being positioned in (b) and geometry-fixed. Finally, the three patches are merged and the pocket region geometry is shown in (c).

patches with $r = 1$ (yellow, blue). How those patches look in the molecule, is shown in Figure 5.3. In the combination step, the patches are placed according to selected overlap triangles first. These triangles are marked in light red in the middle step of Figure 4.7.

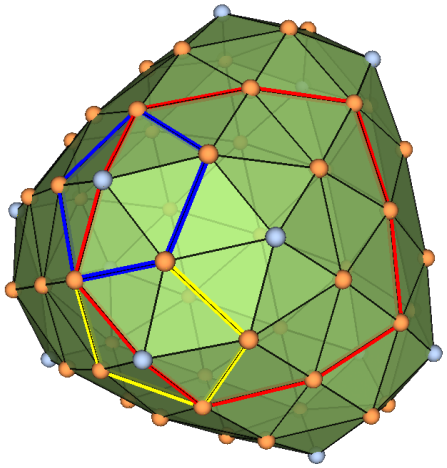


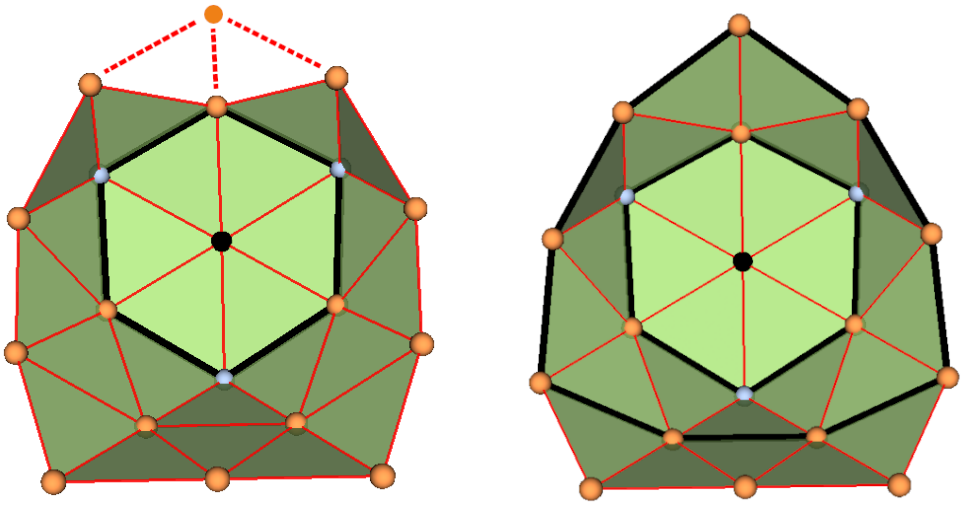
Figure 5.3.: $C_{120}-T$ molecule in dual representation. The three patches that combine to a pocket region in 9.1, are marked.

After this patch placement, the patches can not be merged into the pocket region yet, as there is a geometry flaw: The two nodes in the black circle have the same index in the bond graph, i.e. they are the same node after the merge. Therefore their positions need to coincide. This means that the position of the patches needs to be adjusted in a *geometry fixing*, and then the merge can happen. After all of that, the raw shape pocket region is done, it is now ready to be brought into cone shape. The details of the combination step, especially concerning types of merges and methods for geometry fixing, are discussed in chapter 9.

The Growing Step

The last crucial step of the method is the Growing step. Here, the raw pocket regions generated in the combination step need to be brought to cone shape, meaning that a center for the patch is selected and then layers are built around that center. After that, there has to be an option to add layers to a pocket region if needed. The current state of the implementation is that single hexagon nodes can be added to the pocket region, if three neighbour nodes' positions have already been clearly decided. In the $C_{120}-T$ example, that is just a single node. Figure 5.4 shows the raw pocket region which was the end result in the previous step on the left side. A center node has been selected, colored in black. Around it the first layer is also indicated in black. To complete the second layer, one node needs to be added. Three of its neighbours are already in the embedding, so it's position can be determined. The three edges, which provide three quadratic equations for the position of the new node, are indicated in dotted red lines.

Computing the position of the node yields the right embedding in Figure 5.4, and the second layer can be constructed. However, there are still three nodes on the bottom end



(a) (b)

Figure 5.4.: Bringing the pocket region from Figure 5.3 to cone shape. A single node is missing, its position can be determined from three existing neighbours (a). Afterwards, the second layer is completed (b).

of that pocket region, which are not in a layer. Therefore the cone shape is not reached and one more layer needs to be constructed. This can not be solved by computing the position for one node at a time, as none of those three nodes has three fixed neighbours. A solution is needed to construct a whole layer at a time. This is a work-in-progress though, currently there exists no implementation, but a concept for constructing the coordinates of whole layers. Chapter 10 discusses the challenges and concepts around growing pocket regions, as well as the computation of single node positions, given that there can be obtained three equations.

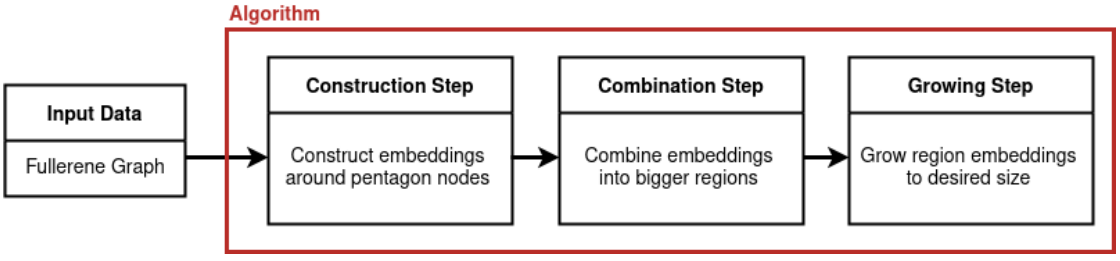


Figure 5.5.: Overview of the main algorithm steps.

Summary

In summary, this algorithm consists of three main steps. Their names and purposes are shown once more in Figure 5.5. The flow of the algorithm is very linear, many steps

have to be executed after each other. Chapters 8 to 10 describe the whole process from beginning to end.

6. Definition of Data Structures

The beginning of this chapter marks the start of the more technical part of this thesis. Now that the theoretical background is summarized and the algorithm concept is worked out, the next focus point is the implementation of the necessary processes. The following chapters will mix a mathematical style of description - equations etc. - with a more implementation-focused one, including some snippets of code. I chose to use object-oriented programming for this framework, simply because there are so many interconnected kinds of information that it would be very hard to keep a good structure otherwise. But that also means that many kinds of information are collected in a few object classes. Introducing these will be the main point of this chapter.

6.1. Patches and Pocket Regions

Patches and pocket regions have already been defined as geometrical objects in chapter 4. In this chapter, they will also be defined as the two big data structures used to hold all the necessary information, except for the bond graph, which is global information and can not be attributed to a single patch or pocket region. Patches are generated first, this is mainly what happens in the construction step of the algorithm (see Figure 5.5), whereas pocket regions are created by combining patches in the combination step and grown in the growing step. A patch is initialized as instance of the object class `Patch()`, the structure of the object class is shown in Table 8.2.

Patch			
Attribute	Dimension	Type	Description
positions	$n_\mathrm{f} \times 3$	float	Node positions
global_index	n_f	int	Global index of each node
triangles	$n_\mathrm{tri} \times 3$	int	Triangles in patch indexing
dual_neighbours	$n_\mathrm{f} \times 6$	int	Sparse adjacency matrix of dual graph
rigid	n_f	bool	Boolean array to mark fixed nodes
Method	Description		
addLayers	Constructs additional layers around minimal size patch		

Table 6.1.: Class diagram of the *patch* class.

Patch Attributes

We will now tentatively address all attributes (and also the methods) of the class. How the data is generated that is saved in these attributes, is a topic for a later chapter, the interest lies in the attributes' purpose as data structures right now. There are four relevant attributes.

`positions` is the array that holds the coordinates of all nodes in the patch. At the same time, it defines the patch index layer: If the patch has n_f nodes and a specific node's coordinates are saved in `positions[u]`, then this node will be referred to as u in the framework of the patch.

`global_index` is an array that ensures identification of the patch indices with the global index layer. This is necessary not only for knowing, which patch node corresponds to which graph node but also to identify patch nodes with pocket region nodes. The index layers are adressed in-depth in chapter 6.2, where the purpose of this array will become more clear. The shape of the array is $N \times 1$, and each node u in the patch index layer gets an equivalent value `U = patch.global_index[u]` corresponding to its index in the global index layer that describes the whole molecule and is the input data to the algorithm.

Then there are `triangles` and `dual_neighbours`. They are the standard graph data structures used throughout this project, identifying the triangles and the edges of the graph structure of the patch.

The last attribute is an $N \times 1$ boolean array called *rigid*, true for each node that is fully determined by other fixed nodes in the region. To understand why this is necessary, several other things have to be worked out first, rigid nodes will be addressed again in the chapters 8.3 and 9.4 for a more detailed description of this attribute's purpose.

Pocket Region Attributes

As definition 4.3.1 states, a pocket region is an embedding of a Fullerene's surface manifold region. It combines multiple patches to a single embedding coherently. Therefore, one can find all attributes that patches also have in a pocket region object. The class diagram is shown in Table 6.2.

The first five attributes are equivalent to the patch attributes. Hereby, `triangles` and `dual_neighbours` hold the graph structure, `positions` holds the node coordinates and `global_index` identifies the pocket region index layer with the global index layer. The purpose of `PocketRegion.rigid` is equivalent to the one of `Patch.rigid` and therefore explained in detail in the chapters 8.3 and 9.4.

The last attribute of the pocket region is a list called `patches`, which simply holds references to all the patches that are part of the pocket region.

Pocket Region			
Attribute	Dimension	Type	Description
positions	$n_\mathrm{f} \times 3$	float	Node positions
global_index	n_f	int	Global index of each node
triangles	$n_\mathrm{tri} \times 3$	int	Triangles in pocket region indexing
dual_neighbours	$n_\mathrm{f} \times 6$	int	Sparse adjacency matrix of dual graph
rigid	n_f	bool	Boolean array to mark fixed node positions
patches	n_patches	patch	List of patches in the pocket region
Method	**Description**		
addPatches	Adds one ore multiple *patch* objects to the patches attribute		
positionByFace	Initial patch positioning for merging		
fixGeometry	Patch position adjustment to fix geometry flaws for merging		
mergeFaces	Merges a patch into the pocket region.		

Table 6.2.: Class diagram of the `PocketRegion` class.

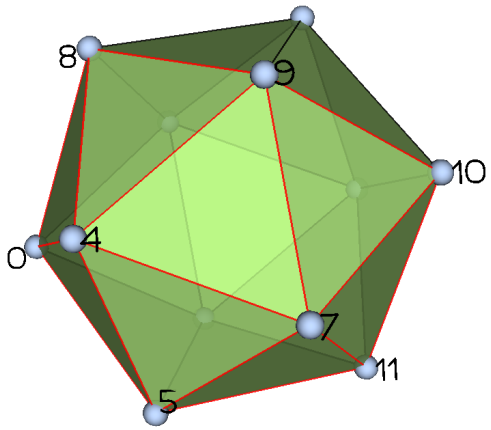
6.2. Indexing

Counting the bond graph, pocket regions and patches, there are three different layers of indexing. The latter two are layers of description for the three-dimensional representation of the molecule, the bond graph serves as global index layer and "construction manual" for the local embeddings. In the tables 6.1 and 6.2, the attributes `PocketRegion.global_index` and `Patch.global_index` are defined. These arrays link each of the indices in a pocket region or patch to the global layer.

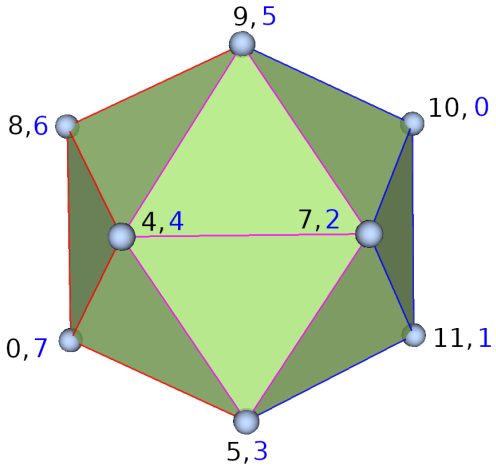
Take for example the smallest Fullerene, C_{20}–I_h. Its dual embedding is shown in Figure 6.1a. Let us now take a look at an arbitrary pocket region from that molecule, indicated by red edges in the same figure. The black indices are global indices from the bond graph.

The pocket region is shown again in 6.1b. The black indices are the same as before, only now there is a set of blue indices from 0 to 7 for the eight nodes of the pocket region. This is the pocket region's index layer, which is used inside the `PocketRegion` class for identification. It is connected to the global index layer by the `PocketRegion.global_index` array, which is in this case `[10,11,7,5,4,9,8,0]`.

The pocket region consists of two minimal size patches. The edges of these patches are indicated red and blue in Figure 6.1b, where the overlapping edges of both patches are magenta colored. These patches are described with patch layer indices inside the `Patch` objects. These indices are indicated in red in Figure 6.2. As in the pocket region, the patch objects' attribute `Patch.global_index` connectes the red indices to the black global

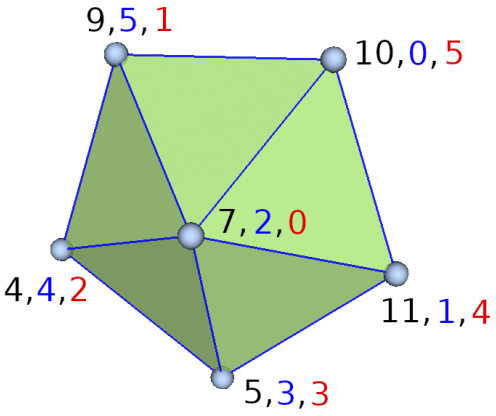


(a) $C_{20}-I_h$ molecule with red pocket region. The black indices are global indices from the bond graph

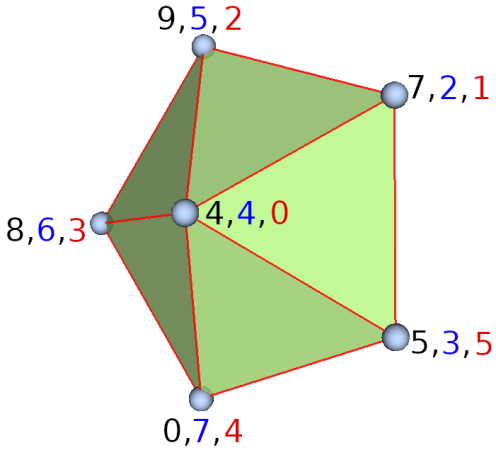


(b) The same pocket region as before. This time the colouring of the edges indicates two patches: One red, one blue, the magenta edges are overlapping edges of both patches. The blue indices are from the pocket region index layer.

Figure 6.1.: Global index layer (black) and pocket region index layer (blue) visualized on a $C_{20}-I_h$ dual embedding.



(a) The blue patch from 6.1b, with patch layer indices in red.



(b) The red patch from 6.1b, with patch layer indices in red.

Figure 6.2.: Both patches from the pocket region in Figure 6.1b. The global (black) and pocket layer (blue) indices are still the same, only red patch layer indices have been added.

layer. These arrays are `[7,9,4,5,11]` (blue patch) and `[4,7,9,8,0,5]` (red patch) for the current example.

This structure means that when searching for overlap faces of, for example, a pocket region and a patch, the indices are compared in the global layer with the help of the `global_index` arrays and then the overlap nodes can be transferred back to the patch and pocket region layers to access coordinates, faces, etc., which are all saved in the local index layers.

Saving the Bond Graph

Because the bond graph (global index layer) is global, it is not suitable to save it inside a pocket region or patch object. Therefore, a new object class is defined.

```
1 class constructionManager():
2     def __init__(self, triangles, dual_neighbours, name=''):
3         self.triangles = triangles
4         self.dual_neighbours = dual_neighbours
5         self.name = name
6         self.pockets = []
```

Listing 6.1: constructionManager class constructor. The original can be found in the file controller.py (see chapter 1.3).

To hold the graph information from the input data, it has the known attributes *self.triangles* and *self.dual_neighbours*. The class uses this information to create an index identification systems that is always coherent. Earlier in this chapter, when the data structures `Patch` and `PocketRegion` were defined, the `global_index` arrays could not be filled, because the bond graph information was not available anywhere in that framework. The construction manager can do this task. That also makes this class the best option for automatizing the generation of patches and pocket regions, such that all pocket regions of a molecule can be generated automatically. On the way, the manager class can chain together all the necessary steps, keep track of important information like the pocket region grouping or the overlaps and transfer it between different parts of the code, and also keep the access to all pocket regions easy by simply saving them in the list `constructionManager.pockets`.

7. Universal Methods

This chapter explains some methods that are widely used throughout the algorithm and can not be attributed to a single process.

7.1. Measuring Distances and Identifying Layers

There exist types of algorithms to compute distances in graphs. In this case, it is rather easy because all edges are of unit length and the graph is planar. Distance can be established by finding a path between two nodes that requires the minimum amount of edges to be traversed.

Because the graph is planar and the manifold has spherical topology, a function that calculates layers of nodes around a single vertex in a circular fashion, can be used as metric. The distance between two points is then calculated by drawing circles around one of the points and seeing, which one the second point is in. This is similar to the construction method itself, which also constructs patches in what one could call circles or layers (a concept sketch of this was shown in Figure 4.2). A function that can calculate layers characterized by their distance d to a starting point can be used for distance measurement in the discrete Fullerene surface manifold, but is also important for the patch building process, specifically for building a working index identification.

This function can be built from the principle "The next layer consists of all nodes, which can be reached by traversing one edge from any node in the current layer. This excludes nodes that have already been assigned to a layer."

In all current applications, the starting layer is a single node. In that case, the next layer will just be all neighbour nodes. This situation is shown in Figure 7.1, where the red edges indicate the first layer that can be constructed from a single node in their middle. The bigger challenge is to continue with the next layer. Assume that we want to use the principle we defined on the green node in Figure 7.1. This yields three new nodes for the next layer, indicated by the green arrows. In order to select the correct nodes automatically, the function can use the ordering of the `dual_neighbours` array. Assume that the surface manifold region in Figure 7.1 is drawn from the inside of the molecule. Then the current layer indicated in red has clockwise ordering, because the neighbours of the center node are organized in a sparse adjacency matrix with clockwise ordering (this was addressed in chapter 2.2). This makes it possible to select a left and right neighbour of the green node in the current layer, indicated with the blue letters L and R in Figure

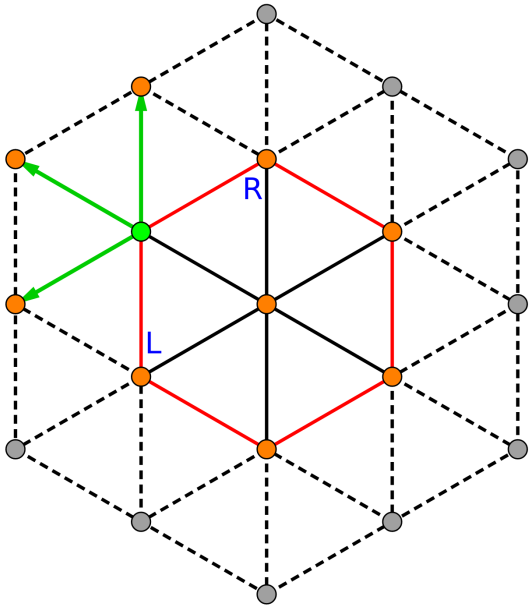


Figure 7.1.: Illustration of one iteration of the nextLayer function. From selected node (green) in the current layer (red), we take all neighbour nodes, which are between the left neighbour node (L) and the right neighbour node (R) in the current layer.

7.1. Because the neighbours of the green node are also in clockwise order in the sparse adjacency matrix, one can select all nodes between L and R, that is when passing L, one starts selecting the nodes and when reaching R, one stops.

Then it only has to be made sure that the layer that we are constructing is in clockwise order. That is already implied, given that the candidate nodes stemming from a single node in the red layer are in clockwise order. Then one can just go around the current (red) layer in clockwise order and select each node's neighbours between the respective L and R nodes to obtain a fully clockwise ordering for the new layer.

```
def nextLayer(dualNeighbors, curLayer):
    potNodes = []

    if curLayer.shape[0] == 1:
        nodes = dualNeighbors[curLayer[0]]
        nodes = nodes[nodes != -1]
        return nodes

    else:
        for i in range(curLayer.shape[0]):
            c = dualNeighbors[curLayer[i]]

            l = curLayer[(i-1)%curLayer.shape[0]]
            r = curLayer[(i+1)%curLayer.shape[0]]
```

```
15
16              #Dead end protection
17              if not l in c or not r in c:
18                  continue
19
20              #Circle through neighbors
21              j = 0
22              append = False
23
24              while True:
25                  if append and c[j%6] == r:
26                      break
27                  elif c[j%6] == l:
28                      append = True
29                  elif append:
30                      potNodes.append(c[j%6])
31
32                  j += 1
33
34          #Select unique entries
35          nodes = []
36          for x in potNodes:
37              if not x in nodes and not x in curLayer:
38                  nodes.append(x)
39
40          nodes = np.array(nodes)
41          nodes = nodes[nodes >= 0]
42
43          return nodes
```

Listing 7.1: Essential parts of the `nextLayer`. The original can be found in the file functions.py (see chapter 1.3).

Listing 7.1 shows the essential parts of the corresponding function `nextLayer`, which implements that methid. First, note the inputs: only the sparse adjacency matrix of the surface manifold (region) and the current layer are needed. Line 4 to 7 are executed, if the current layer is a single node. In this case, the next layer are simply all the neighbours, as it was also described in connection to Figure 7.1.

From line 9, the more general case for any closed layer is executed. The `for` loop in line 10 runs over all nodes in the current layer, in clockwise order when seen from the inside of the molecule. When a node is selected, its neighbours are saved in `c`, R and L are defined as the next and previous node in the current layer (we can verify that this is correct by looking at the example in Figure 7.1 and remembering the clockwise ordering). Then the candidate nodes for the next layer (the "green arrowed" nodes) have to be found. For that, the function circles through c in line 24. When L is reached, the second condition is fulfilled and the *append* parameter is set to *True*. From here on, the third condition is fulfilled in every iteration and the entries of c are filled into the *potNodes* array that holds the candidate nodes. This happens, until R is reached. Then the first condition is fulfilled and the loop is stopped.

Many nodes get selected multiple times in this process, simply because they are neighbours of multiple nodes in *curLayer*. Therefore, one selects the unique entries in *potNodes* in lines 35 to 38. As a last manipulation, the -1 entries are filtered out in line 41, as it was discussed in chapter 2.2.

7.2. Patch and Pocket Region Movement

Patches and pocket regions need to be repositioned from time to time. To reach any possible position in three-dimensional space, it is necessary to code two operations: translations and rotations. The original functions can be found in the file functions.py (see chapter 1.3).

Translations

The way these operations are coded, they require two inputs: the object that is to be translated and a vector that represents the translation. The only condition for the object is that it has a *object.positions* attribute, which both patches and pocket regions have.

```
1 def translate(obj, tVec):
2     obj.positions[:,0] = obj.positions[:, 0] + tVec[0]
3     obj.positions[:,1] = obj.positions[:, 1] + tVec[1]
4     obj.positions[:,2] = obj.positions[:, 2] + tVec[2]
```

Listing 7.2: The *translate* function for patches and pocket regions.

With these initial conditions fulfilled, each of the node positions simply gets shifted by the translation vector.

Rotations

To rotate a whole patch or pocket region, a few more inputs are needed. The system is the same, so a object to rotate is one of the inputs. Then, a point to rotate around, a rotation vector for the direction of rotation and a rotation angle are needed to complete the inputs.

```
1  def rotate(obj, rVec, refP, angle):
2      #first, translate grid such that refP is origin
3      translate(obj, -refP)ting
4
5      #rotate around origin
6      for i in range(obj.nodes.shape[0]):
7          obj.nodes[i] = rotateVector(obj.nodes[i], rVec, angle)
8
9      #translate back
10     translate(obj, refP)
```

Listing 7.3: The *rotate* function for patches and pocket regions

The function first translates the reference point of rotation to the origin. Then all node coordinates can be treated as vectors and rotated around the rotation vector by the specified angle, which is executed by the *rotateVector* function in line 7 of Listing 7.3. After all rotations are completed, the sytem is translated back to its original position.

7.3. Generating Dual Neighbours From Triangles

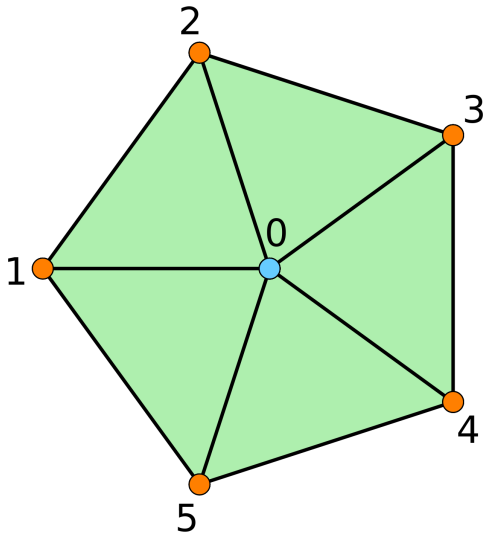
The bond graph of the whole molecule, but also the graphs of patches and pocket regions are always kept in two different data structures: the `dual_neighbours` array and the `triangles` array. Generally the program updates triangles during operations on patches and pocket regions, but doesn't update the dual neighbours. Instead the dual neighbours can always be generated from the updated faces array. The function computing the dual neighbours is called `generateDualNeighbours`.

```
1  def generateDualNeighbours(triangles):
2      nMax = np.amax(triangles) + 1
3
4      dual_neighbours = np.ones((nMax, 6), dtype=int) * -1
5
6      for i in range(nMax):
7          nb = neighbours(i, triangles)
8
9          for j in range(nb.shape[0]):
10             dual_neighbours[i, j] = nb[j]
11
12     return dual_neighbours
```

Listing 7.4: The *generateDualNeighbours* function that computes the dual neighbours from the faces

The function is printed in Listing 7.4. It starts out in line 2 by computing the amount of nodes n_{Max} in the `triangles` array. Then the `dual_neighbours` array is defined with the shape $n_{Max} \times 6$ to hold six neighbours for each node. All values of this array are set to -1. This way, pentagon nodes automatically have -1 as their last neighbour node. A little reminder about that: basically -1 is easy to filter out when using the `dual_neighbours` array for computations. It is also noteworthy that this array does not describe a full molecule in the vast majority of cases, but rather a patch or pocket region, which have an outer boundary. Boundary nodes will also have multiple entries of the value -1, where their neighbours are not specified.

In the lines 6 to 10 of Listing 7.4, the neighbour values for each node are calculated and filled into the `dual_neighbours` matrix. The function `neighbours` is the center piece of this process and deserves a closer look. It is easiest to explain how this function works with a simple example: A minimal size patch with a center pentagon node and five nodes around it. The shape of the embedding is for example shown in Figure 4.1a and a sketch on the index assignments of the nodes as well as the corresponding `triangles` array is provided in Figure 8.3.



$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 0 & 2 \\ 0 & 3 & 4 \\ 4 & 5 & 0 \\ 1 & 0 & 5 \end{pmatrix} \tag{7.1}$$

Figure 7.2.: Index assignment for embedding of pentagon node and neighbours (left) with the corresponding *triangles* array in matrix form (right).

`Neighbours`takes a single `vertex` and the complete `triangles` array as input. In line 3, `triangles` is filtered, only faces containing the single node from the input are of interest. In the example, let's will choose 0 as `vertex`. Then the filter keeps all faces, because each face has an entry of value 0.

In line 6 to 8, the `np.roll` function is used to bring *vertex* to the first position in the row. `np.roll` shifts the index of every entry of a one-dimensional array by a specified integer number. Arrays can be rolled "to the right" using positive integers and "to the left" using negative integers. An entry on the last index of the array is taken to the first index when being rolled to the right and vice versa. Take for example one row from the SAM in Figure 7.2:

$$\begin{pmatrix} 3 & 0 & 2 \end{pmatrix} \overset{np.roll(1)}{\rightarrow} \begin{pmatrix} 2 & 3 & 0 \end{pmatrix} \overset{np.roll(-2)}{\rightarrow} \begin{pmatrix} 0 & 2 & 3 \end{pmatrix} \tag{7.2}$$

Equation (7.2) shows two example operations for rolling a row, the first one rolling "to the right" by one and the second one rolling "to the left" by two. The reason for using this function is that, other than simple permutations, it preserves the ordering of the faces it is applied to.

```
1 def neighbours(vertex, faces):
2     #choose all faces that vertex is a part of
3     nb = faces[np.sum(faces == vertex, axis=1) > 0]
4 
5     #roll the vertex to pos 0 in the row
6     for k in range(nb.shape[0]):
7         n = np.where(nb[k] == vertex)[0][0]
8         nb[k] = np.roll(nb[k], -n)
```

```
9
10      #take the edges that do not contain 0
11      linkEdges = []
12      for i in range(nb.shape[0]):
13          linkEdges.append([nb[i,1], nb[i,2]])
14
15      linkEdges = np.array(linkEdges)
```

Listing 7.5: The *neighbours* function returns the dual neighbours of a single input vertex.

In line 7 of Listing 7.5, the index of the `vertex` entry is found and in line 8, that index is used to roll the array to the left, until `vertex` is the first entry in the row. After this step, the example matrix from Figure 8.3 will look like this:

$$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 3 \\ 0 & 3 & 4 \\ 0 & 4 & 5 \\ 0 & 5 & 1 \end{pmatrix} \tag{7.3}$$

From every row in this matrix, the edges that do not contain `vertex` are selected and saved in the `linkEdges` list. This happens in the lines 11 to 13. Going back to the example again, these edges can be written in matrix form like this:

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 1 \end{pmatrix} \tag{7.4}$$

Searching for these edges in Figure 8.3, you see that these are all edges that contain direct neighbours of the `vertex` 0. Additionally, they are all oriented in clockwise order. This is crucial for keeping the additional ordering information in the sparse adjacency matrix for the dual neighbours. The only challenge left is to extract the unique neighbour nodes in the correct order from the list of edges.

When beginning with an arbitrary edge, for example $(2, 3)$, 2 and 3 can be added to the list of neighbours of 0. Thereby, 3 is the "end node". To find the next node in order, one simply has to find the edge that starts at 3, in this case $(3, 4)$. Then 4 can also be added to the neighbours. In the end, the full array of neighbours reads $(2, 3, 4, 5, 1)$, which is a representation with correct ordering.

Now the example that I chose was a very basic case. The edges are building a closed loop and are already well-ordered. The `neighbours` function is also prepared for vertices, whose neighbours don't build a closed loop. This is for example the case for vertices on the boundary of the embedding. Take for example 5 as `vertex` of the `neighbour` function

in the embedding from Figure 8.3. Then the matrix of edges is

$$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \tag{7.5}$$

and the neighbours in correct ordering are $(1, 0, 4)$. The process of selecting neighbour nodes from the edges is handled by the `neighbour` function in its entirety, it is quite lengthy to analyse the code though, so I refer the interested reader to the original code documents for the second part of that function, they can be found in the file functions.py (see chapter 1.3).

In the end, the `neighbours` function will always return an array of all neighbour nodes that it found to the input `vertex`. This array is called `nb` in the function `generateDualNeighbours` and used to fill the `dual_neighbours` SAM in lines 9 and 10 of Listing 7.4. With that, the generation of dual neighbours from the triangles of an embedding is finished and the SAM can be returned.

8. The Construction Step

The first step of the sub-divided algorithm is the construction step. This step is all about the `patch` objects. Before constructing the patches themselves (i.e. the embeddings), it is necessary to know the size of each patch, such that patches do not have to be grown after the initial construction process. How big a patch has to be, depends greatly on the distance of the pentagon node in the patch's center to the closest other pentagon node. Computing the right size for each patch requires two things:

1. It has to be decided, which patches are grouped into a pocket region and therefore need to be overlapped

2. The patch radii inside a pocket region have to be distributed such that the influence of pentagon nodes on the pocket region's shape is as even as possible.

Only after this can the instances of the `patch` class be generated.

Figure 8.1 shows the complete overview of the construction step. The output of this part of the algorithm consists of two things: first, there are the patch objects in correct size, and then there are the pocket region groups, which come from the grouping algorithm. These are just lists holding the information about which patch belongs to which pocket region, which will be needed again when instantiating pocket regions.

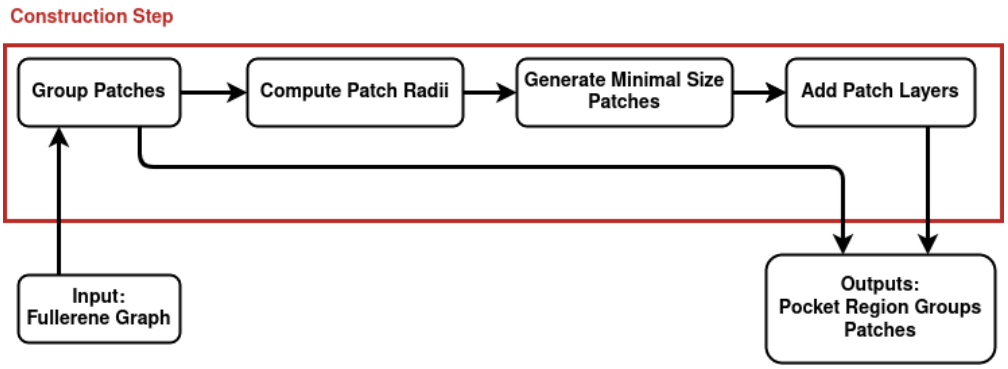


Figure 8.1.: Construction step algorithm overview.

8.1. Grouping Pentagon nodes

As first real step of the algorithm, the sparse adjacency matrix can be used to determine, which pentagon nodes should be combined to a pocket region. This can be a little tricky to do in some cases, as Fullerenes can have vastly different distributions of pentagons. One of the more difficult cases is for example the $C_{60}-I_h$ Buckyball (see Figure 1.1), which has a totally even distribution of pentagons. In this case the desirable outcome of the algorithm is two pocket regions with 6 pentagon nodes each. One would want the same outcome for a carbon-nanotube-like molecule like the $C_{360}-D_{5H}$ (see Figure 1.3), though in this case it is much easier to decide, because there exist two clearly separated groups of six pentagons, one in each end of the nanotube.

The Distance Matrix

To sort the twelve pentagon nodes in a Fullerene dual surface manifold into groups, we simply compare their distances to each other. For that purpose, a distance matrix D is of dimension 12×12 is defined. The matrix entry $D_{u,v}$ holds the distance between pentagon node u and pentagon node v. u and v are integer numbers, as in the bond graph, the distance between u and v is defined by the minimum amount of edges that have to be traversed to get from u to v. In chapter 7.1, it was demonstrated how to measure this distance in a Fullerene surface manifold using the `nextLayer` function. With this function, a distance matrix can easily be computed by scanning the whole molecule for pentagon nodes, starting from a single pentagon node, and then filling up the rows of the matrix.

Implementation

The grouping mechanism is quite a rudimentary piece of code. This should definitely be replaced in the future by a more sophisticated method, for example a hierarchical clustering based on the same distance matrix. For the purpose of this thesis, I stuck with what worked for me, simply because there were more pressing problems to solve. The process is handled by a function called `groupPtgNodes` and printed in listing 8.1. It takes the distance matrix defined above and the sparse adjacency matrix of the surface manifold as inputs.

```
1 def groupPtgNodes(dist, dual_neighbours):
2     groups = []
3     active = np.ones(12, dtype=bool)
4 
5     #Step 0: Decide on where to start: on the nodes with minimal distance
          connections.
6     distMin = np.amin(dist[dist > 0])
7     current = np.any(dist == distMin, axis=1)
8 
```

```
9       #Step 0.5: Order nodes by minDist connections
10      order = orderByMinCount(dist)
11
12
13      #Step 1 : Get initial disjoint groups from ordered indices
14      for index in order:
15          #Take only nodes that we currently want to look at
16          if not current[index] == True:
17              continue
18
19          group = nodeCluster(index, dual_neighbours, distMin)
20
21          if np.sum(active[group]) == group.shape[0]:
22              #Now append the group and set the indices inactive
23              groups.append(group)
24
25              for x in group:
26                  active[x] = False
27
28
29      #Step 2: Take care of leftover active indices
30      groups = undecidedIndices(groups, dist, active)
31
32      #Step 3: order the groups from center to outer nodes
33      groups = sortGroups(groups, dist)
34
35      return groups
```

Listing 8.1: The function that groups the pentagon nodes and therefore decides about the configuration of pocket regions. The original can be found in the file functions.py (see chapter 1.3).

The first two parameters are the list `groups`, which holds the current groups, as well as `active`, a boolean array which holds information about which nodes are grouped and which aren't (if a node is active, it is not grouped).

The most important parameter in this function is called `distMin` and defined in line 6. It represents the minimal distance between any two pentagon nodes in the manifold. Wherever this minimal distance between two nodes appears, it is pretty clear that they need to be grouped together. The advantage is that many Fullerenes have a clear minimal distance that will appear for many pentagon node pairs.

It is often predictable what the minimal distance between pentagon nodes will be. A coarse subdivision is possible between Fullerenes that have minimum distance of 1, and those with minimum distance of at least 2. A distance of 1 means that the molecule has adjacent pentagon faces in the cubic representation. Generally, it has been found that Fullerenes with isolated pentagons (a minimal distance $d \geq 2$) are more stable. This is referred to as the *Isolated Pentagon Rule* (IPR). An IPR-Fullerene is a Fullerene with a minimal distance between pentagon nodes of larger than one in the dual representation.

If it is a non-IPR Fullerene, the minimal distance can be 1. Then this distance will also

frequently appear. For small IPR Fullerenes, the minimal distance is two, we will see that frequently as well. The boolean array `current` marks all nodes that are in minimal distance to at least one other node. The array `order` holds the indices of all nodes, ordered from a high amount of minimal distance neighbours to the lowest amount of minimal distance neighbours.

The crucial part of the function begins in line 14. Step one is to create disjoint groups of only minimal distance neighbours. The if-statement in line 16 only comes into effect, when the algorithm gets to the point where it looks at nodes that don't have any minimal distance neighbours, so by then step 1 is effectively finished. In line 19, the function `nodeCluster` generates the initial groups. The way this works is that starting with the node `index` that was selected from `order`, all minimal distance neighbours of the selected node are added to the group. Afterwards, the minimal distance neighbours of all new nodes in the group are also added to the group until no new minimal distance neighbours are found.

After the `order` array, each node is selected and a grouped with its minimal distance neighbours by `nodeCluster`. If this group is disjoint to all other existing groups, it will be added to `groups` and all contained indices set inactive, such that they can not be in another group.

A little example can help understand this function better. Figure 8.2 shows a cluster of pentagon nodes, where the minimal distance neighbours are marked with dashed lines between them. Running `groupPtgNodes` on this cluster will put the green node first in `order`, because it has most minimal distance neighbours. For the this node, the group as

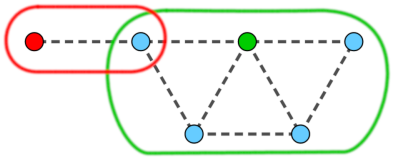


Figure 8.2.: Minimal distance neighbours of a group of pentagon nodes indicated by the dotted lines. In the first step of `groupPtgNodes`, the marked groups are selected for the red and green node. Because the red node has the higher number of minimal distance neighbours, the group of the green node will not be added to `groups` and the green node is handled in step 2 of the function

indicated by the green loop will be selected and added to `groups`. All nodes nodes in the group are set inactive. The only node left ungrouped is the red node. It is grouped with its only minimal distance neighbour, but that neighbour is inactive and therefore this group is not added to `groups`. The red node is then handled in step two of `groupPtgNodes`.

In general, there will be pentagon nodes in a molecule that do not have any minimal distance neighbours. These nodes will not be grouped during step 1, therefore the algorithm takes a look at them during step 2. For each ungrouped node, it has to be

decided if the node should start a new group or if it should be added to an existing group. This is the more complicated part of the algorithm, because individual decisions need to be taken for every situation. I went with a little bit of a trial-and-error approach to find a method that works for as many molecules as possible. In the current implementation, an ungrouped node is assigned to an existing group, if the following conditions are fulfilled.

1. The group has the smallest distance to the node (measured by average distance of group members).
2. The group contains at maximum five nodes.
3. At least one of the group member nodes is a closest neighbour[1] to the ungrouped node.
4. The average distance from group members to ungrouped node does not exceed 30% of the maximum distance of any two pentagon nodes in the surface manifold.

All four of these conditions serve a purpose. Condition one is to ensure that no other group is closer, condition three ensures that no other node is closer. Condition two makes sure that a group cannot grow to big. And condition four is a decider for when a single node is too far away from all other groups to be added to any.

As a last step, the nodes are sorted by average distance to other nodes in their group. This will come in handy later, when for each group a center node has to be selected, and the average distance to all other group members is a good parameter to decide, how central a node is placed. More about this later in chapter 9.3.

8.2. Determining Patch Radii

When the pentagon nodes are grouped, their radii can be computed. There are several conditions that one has to think about when doing this, the overall goal is to make sure that patches overlap consistently, such that they can be merged, and that the radii are as even as possible to ensure an even pocket region shape.

Radius Conditions

Pentagon nodes are the shape-defining objects of dual Fullerene surface manifolds. From that it follows that one has to start constructing from all pentagon nodes simultaneously, in order to respect the influence of each pentagon node on the shape of the embedding. But then one also has to make sure that the patch radii are distributed as evenly as possible, that is if there are two patches close to each other, it is necessary to choose the

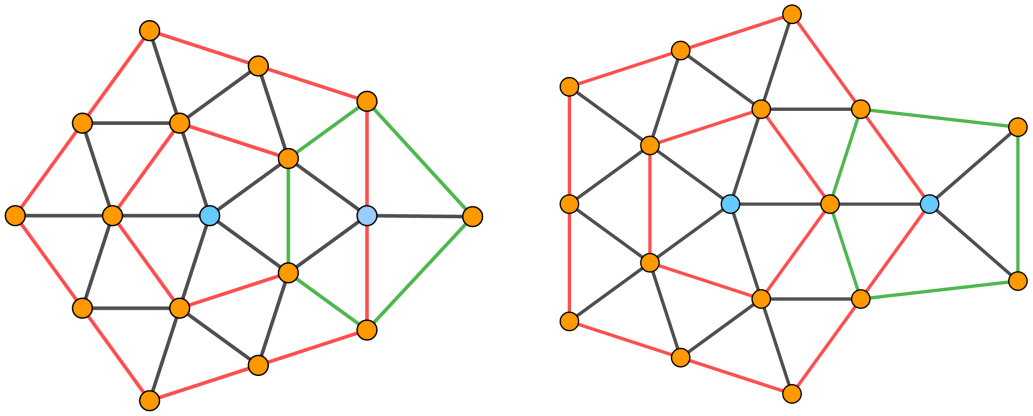
[1] A closest neighbor is not equivalent to a minimal distance neighbour. A minimal distance neighbour is a closest neighbour, with the additional condition that the distance between the nodes is equal to the minimum distance in the distance matrix of pentagon nodes. On the other hand, a clostest neighbour can be in any distance of a node, as long as there is no other node closer.

most even radii possible. Basically it is assumed that the shape a of a surface region region determined by its closest pentagon node.

The radii of patches also have to be big enough to guarantee overlap to other patches in the group. This can be expressed in a simple equation. Assume that two pentagon nodes labelled with indices 1 and 2 have a distance of d. In order to have overlapping triangles, their radii have to fulfill the equation

$$d + 1 = r_1 + r_2. \tag{8.1}$$

The combined radii have to be one bigger than the distance to ensure an overlapping face. An overlapping edge or vertex is not enough, because we need three overlapping nodes to position the patches correctly with respect to each other. An equation of the

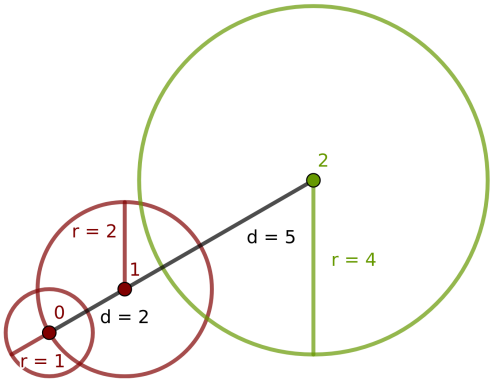


(a) (b)

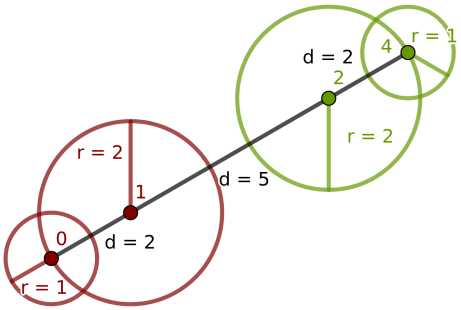
Figure 8.3.: Two examples of overlapping patches (patch layers drawn in red and green), both situations with $d = 2$ between the blue pentagon nodes, $r_{\text{red}} = 2$ and $r_{\text{green}} = 1$, such that equation (8.1) is satisfied. This causes an overlap of 3 (a) and 2 (b) triangles.

form (8.1) can be written down for every pair of pentagon nodes in a pocket region. In theory, this can build a system of equations with $n \in \{2, ..., 6\}$ unknowns $r_1, ..., r_n$ and $(n-1) + (n-2) + ... + 1 = \frac{n \cdot (n-1)}{2}$ equations for n patches in a group.

Every equation of the form (8.1) has two unknowns. One degree of freedom is left in the system and the equation has multiple possible solutions. However, the even radii equation from above selects a solution out of all possible ones, based on how even the radii are. Therefore the radius of a patch is fixed as soon as a single equation based on distance to a neighbour has been written for it.



(a) A very simplified visualization of three patches, built around pentagon nodes 0, 1 and 2. The red patches have radii $r = 1$ and $r = 2$, the green one $r = 4$; the distances between pentagon nodes are indicated with d.



(b) A similar situation to the one in Figure 8.4a, only that there is one more node. This combination of nodes can not overlap and build a pocket region.

Challenges

There are some inconsistencies that arise with this method. How to deal with them can be explained using some very theorized examples. First, in the case of an even d, the two radii r_1 and r_2 added up will have to give an uneven number and the condition to have them as even as possible dictates to choose $r_1 = r_2 + 1$ or $r_1 = r_2 - 1$, two possible solutions. This can be solved by taking a look at how central each of the nodes is.

It is possible to encounter situations like in Figure 8.4a. It features two pentagon nodes with a distance of $d = 2$ to each other (the red nodes) and the green node that has a distance of $d = 5$ to the red node labelled 1. That means that equation (8.1) computes radii of 1 and 2 for the red nodes, no matter which node actually has which radius. On the other hand, computing radii for node 1 (red) and node 2 (green), a radius of 3 is required of both. These two results are cannot be fulfilled at the same time.

In that case, green node needs a bigger radius than what the even radius condition selects. A workaround for this problem is to assign a radius to nodes with a closer nearest neighbour first and then go to the ones that are further away afterwards, when selecting their nearest neighbour, the algorithm can check if there is already a radius assigned to this node. That is, if the red nodes are assigned radii first, equation (8.1) becomes easily solvable for the green node's radius.

Lastly, there are situations that can appear with four or more pentagon nodes in a pocket region, where it will not be possible to find a combination of radii that enables the construction of a connected pocket region. An example is shown in Figure 8.4b. The

only solution to this type of problem is to split up the pentagon node grouping, such that the result is two pocket regions instead of one.

Implementation

The concept worked out above can be implemented with the following structure: First, select a group and from that group, select a node to start with. That will usually be the node with most minimal distance neighbours in the group. Use the minimal distance to compute the radius of the center node from equation (8.1). This also computes the radii of the minimal distance neighbours, which cascades to their neighbours by using equation (8.1) once again. The original function can be found in the file functions.py (see chapter 1.3).

8.3. Generating a Minimal Size Patch

Now, patch construction can be started. The class of patches was already introduced in chapter 6.1, addressing the data structures `patch` objects consist of. Now the focus shall lie on filling the data structures with the fitting information. All attributes will be addressed in the order that they are defined in Listing 8.2. This listing shows the *constructor* of the *patch* class, which constructs object instances. All crucial attributes are generated in this function.

```
1  class Patch():
2  def __init__(self, pos=(0,0,0)):
3      self.positions = fcs.generateDual(pos)
4      self.global_index = np.zeros(self.nodes.shape, dtype=int)
5
6      self.triangles = np.array([[0,1,2],[0,2,3],[0,3,4], [0,4,5],
           [0,5,1]])
7      self.dual_neighbours = fcs.generateDualNeighbours(self.triangles)
8
9      self.rigid = np.zeros(self.nodes.shape[0], dtype=bool)
10     self.rigid[0:6] = True
```

Listing 8.2: Patch class constructor.

The most important attribute is the first one: `self.positions` saves the positions of all nodes in the patch. In the initialization process, every patch is created as minimal size patch. The patch can be increased in size right after the initialization process (The class diagram in table 6.1 shows that the method `addLayers` does exactly that). For now the focus lies on assigning coordinates to the first six nodes. The function `generateDual` creates a minimal size patch, the input parameter `pos` determines the position of the center pentagon node. The patch cone is then built opening towards positive z direction, such that all edges are of unit length. For that, the remaining five nodes are simply

constructed as regular pentagon in a plane parallel to the x-y-plane with z-coordinate $\boldsymbol{x}_{\text{pos}} + \left(0, 0, \sqrt{1 - 1/(4\sin(\frac{\pi}{5})^2)}\right)^T$.

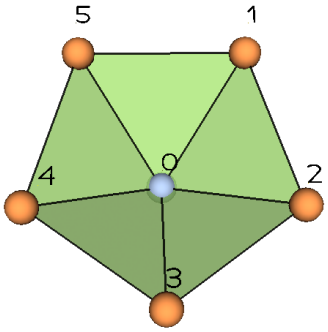


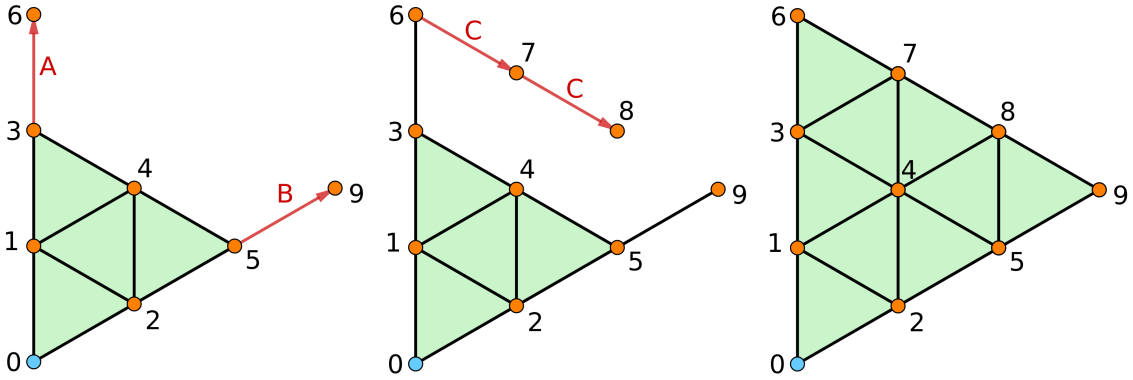
Figure 8.5.: Minimal size patch with indices from positive z-direction

The next attribute in the initialization process in Listing 8.2 is `self.global_index`, which is only being filled with zeros because the reference to the graph information can not be added from inside the class. This has to be handled from the outside, the corresponding processes were addressed in chapter 6. Generally, the `nextLayer` can be used to find the global indices of all layers for the patch.

In the minimal size patch, there are five triangles being saved in the `self.triangles` array. When looking at the patch from positive z (that is, from the inside of the molecule), the nodes that define these triangles are saved in clockwise ordering. The convention is to use counter-clockwise ordering when looking from outside the molecule, which implies the clockwise ordering from inside the molecule. From the triangles, the dual neighbours information can be generated by the `generateDualNeighbours` function that was introduced in chapter 7.

Finally, I will introduce the `rigid` property for nodes in a patch. The background of this property is that it will be necessary to change the shape of patches in the merging process. There are certain nodes in a patch that should not be used for changing the patch shape though. For patches, this is specifically the center node and all other nodes contained in the minimal size patch. This requirement is less due to geometrical reasons, because geometrically it is definitely possible to change the shape of a minimal size patch and still keeping all triangles equilateral. Changing the minimal size patches simply did not seem to be necessary in any case, therefore it would be unnecessarily complicated to do it. However, there showed to be a possible exception to this rule, which still is an unsolved problem and addressed in chapter 11.1.

When merging patches into pocket regions, the `rigid` property is transferred from patch to pocket region and will become gradually more important towards the growing step. For now though, all nodes of the minimal size patch are rigid, while the rest of the standard shape patch is not.



(a) Computing corner nodes. (b) Computing sector nodes (c) Filling up with triangles.

Figure 8.6.: Schematic overview of the three steps taken to add a layer to a sector with two layers.

8.4. Additional Layers

The minimal size patch coordinates are constructed, so now an arbitrary amount of layers is to be added. This can be done by constructing single sectors and then combining the new nodes from all sectors and corners. The structure of layers and sectors was discussed in chapter 4.1.

Growing a Sector

Patches are constructed in standard shape, where the corners are straight lines, which makes it easy to construct the position of new corner nodes. Figure 8.6 shows a sector of radius two, which is grown to size three in three steps. The vectors $\vec{A}$ and $\vec{B}$ defining the new corner nodes can be calculated from the previous corner node positions. In this example, the vectors are computed as

$$\vec{A} = \vec{p_3} - \vec{p_1} \quad \text{and} \quad \vec{B} = \vec{p_5} - \vec{p_2}. \tag{8.2}$$

Hereby, p_n is the position vector of the vertex labelled with n. The new corner node positions are then

$$\vec{p_6} = \vec{p_3} + \vec{A} \quad \text{and} \quad \vec{p_9} = \vec{p_5} + \vec{B} \tag{8.3}$$

When adding a complete layer to a patch, all five sectors have to be calculated, but they share corners, so it is only the case for the first sector that both corner nodes have to be calculated. But that is a detail that the algorithm needs to deal with.

When the corner node positions are computed, it is quite easy to infer the sector node positions. The number of sector nodes in a layer depends on the *layer index* l, which starts from zero. In Figure 8.6, the 0'th layer is the node 0, the first layer are the nodes

1 and 2, etc. Then a rule for the number of sector nodes s in a layer can be put as follows:

$$s = l - 1 \tag{8.4}$$

The logical process behind this is the following: The number of edges in a sector that belong explicitly to one layer is the same as the layer index l. Because a layer is not closed inside a single sector, the total number of nodes in a layer k can be written as $k = l + 1$ (the number of edges plus one). Of those k nodes, there are always two corner nodes and the rest are sector nodes, therefore $s = k - 2$ and equation (8.4) holds.

To get the positions of the s sector nodes, one can once again define a vector C that spans between the two new corner nodes.

$$\vec{C} = \vec{p_9} - \vec{p_6} \tag{8.5}$$

The positions of the sector nodes are then computed by taking

$$\vec{p_i} = \vec{p_6} + i \cdot \vec{C} \quad \forall i \in 1, ..., s \tag{8.6}$$

This is the second step of the whole process and visualized in Figure 8.6b.

With these coordinates, all node positions for the new layer are known. What is missing is the information about new triangles that needs to be added to the `patch.triangles` array. These can be specified as follows: Assume that a new layer of nodes has already been computed. To specify the triangles, it is best to start with the previous layer $l - 1$. That layer has $l - 1$ edges and $l - 2$ sector nodes in a single sector (see equation (8.4) and after). Each of the edges builds a triangle with one of the sector nodes in the new layer, and each of the sector nodes in the old layer builds a triangle with an edge in the new layer. The end result for the example sector is shown in Figure 8.6c. The triangles are constructed as follows: The previous layer is $[3, 4, 5]$, the new layer is $[6, 7, 8, 9]$. For each node in the old layer, one selects a pair of nodes to obtain the triangles $[3, 6, 7]$, $[4, 7, 8]$ and $[5, 8, 9]$. Then, one selects pairs of nodes from the previous layer, and each of those is combined with one of the sector nodes to obtain $[7, 4, 3]$ and $[8, 5, 4]$. Here it is important to keep the ordering clockwise when looking at the sector from inside the molecule.

Implementation

As this is a pretty straightforward implementation, the code is not shown here. Basically all one has to do is compute the five new corner nodes and then the sector nodes for each sector, add them to the patch and check that the corner nodes are not added twice. The original function is called `pocket.addLayers` and can be found in the file classes.py (see chapter 1.3).

9. The Combination Step

Exiting the construction step and entering the combination step, the full patch objects (twelve per molecule) and information on which pocket region they belong to are the current state of computation. The construction step uses the grouping information to merge single patches to pocket regions. An overview of the necessary steps is given in Figure 9.1. The algorithm consists of three main steps. First, a center patch is selected which serves as the first version of the pocket region. Then, adjacent patches are placed by overlapping triangles. This initial placement of patches is not coherent in some situations and geometry flaws can arise. That is when a *geometry fixing* becomes necessary. Finally the merging process takes place and the patch becomes part of the pocket region. These three steps have to take place for each patch, except for the patches that have been selected as center patch of a pocket region.

Generally, one can roughly distinguish between two types of merges, one of which requires geometry fixing, while the other doesn't. Before getting into the details of the three operations required to execute on each patch, taking a look at the more theoretical side of merging is beneficial.

9.1. Pentagon distance and types of merges

In the following section, the basic types of merges for close distances will be worked out. For now, the focus lies on merges with a pentagon node distance of one and two. Merges can be divided into two distinct types, which will be introduced first.

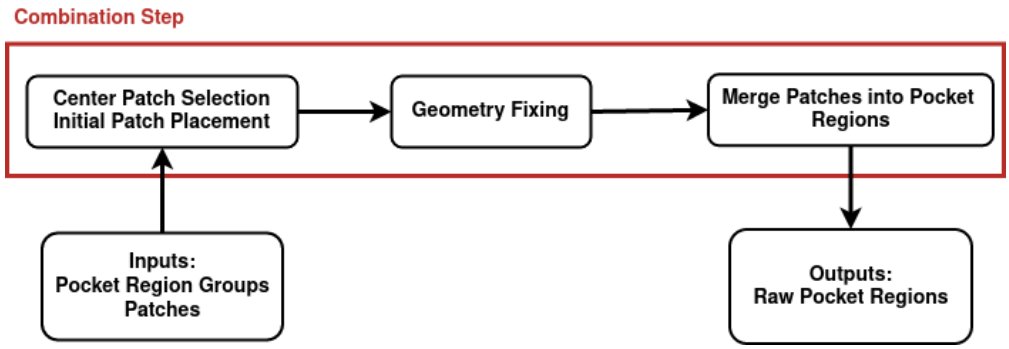


Figure 9.1.: Combination step algorithm overview.

Terminology

First, some terminology and structuring to make the description of the merge types easier. A standard shape patch has a shape that could be described as a wide open cone. In this cone, five sectors can be identified (see chapter 4.1). A sector corresponds to one of the flat regions that emerge from one of the five triangles in the first layer around the pentagon node. The sectors are separated by the corners of the cone.

If all overlap triangles are in the same sector in at least one of the two patches, then the merge is called a *sector merge*. When talking about small and medium sized pocket regions (i.e. patch radii not bigger than two), this type of merge is always associated with a three-face overlap.

On the other hand, when the overlap includes triangles from two sectors for each of the patches, the merge type is *corner merge*. In corner merges, the edges that define the corner in each patch, overlap after the patch positioning process. For all sizes of molecules, a corner merge is associated with an overlap of two triangles in total.

D = 1 merge

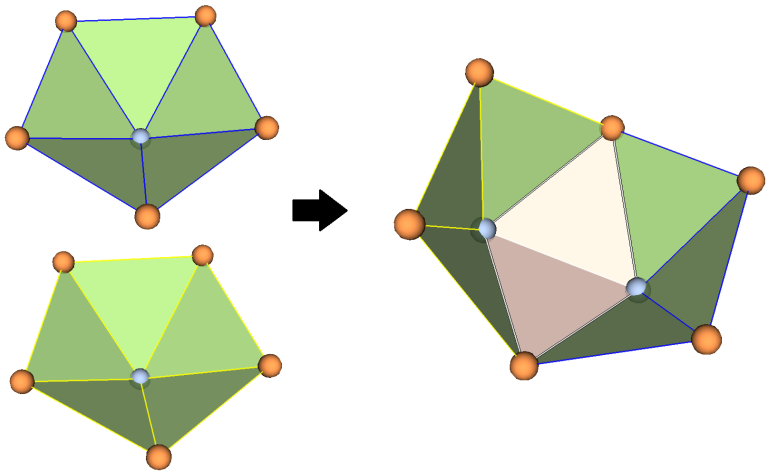


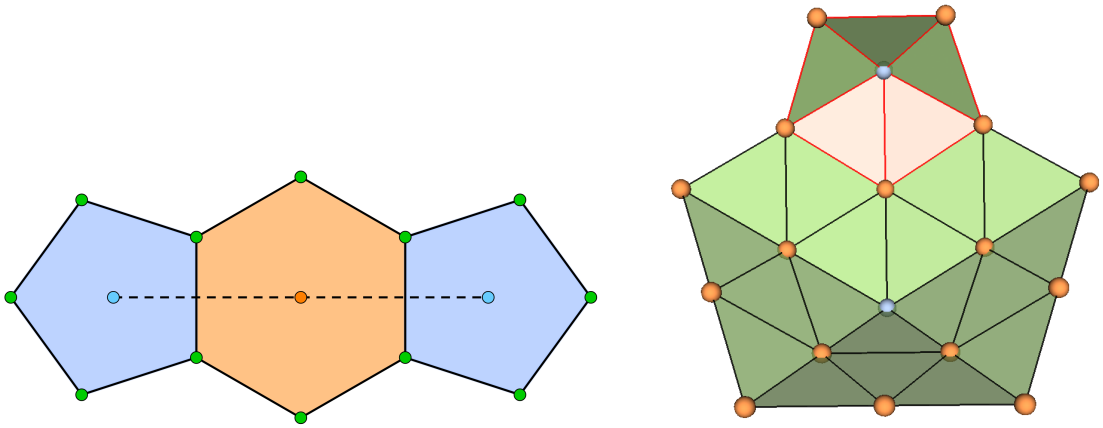
Figure 9.2.: Corner merge of minimal size patches (one blue, one yellow). The overlap triangles are colored in light red.

The first possible case is to merge two minimal size patches. This qualifies as a corner merge, a sector merge does not exist for patches of that size. As the previous description has mentioned, corner merges have two overlapping triangles, which are marked in light color in Figure 9.2. The pocket region in that Figure is also free of geometry flaws. This merge plays an important role in all non-IPR Fullerenes. C_{20}–I_h, the smallest Fullerene, can be completely constructed using this merge. Another example is the pocket regions

of C_{1140}–T_d, which also feature adjacent pentagon faces. Both molecules are shown in Figure 1.1 in the introduction.

D = 2 merges

For $d = 2$, there are two possible cases to look at. Both feature one patch of radius two and one minimal size patch, governed by equation (8.1). The first option is that two pentagons are connected by sharing a single adjacent hexagon face in the cubic representation. This situation is shown in Figure 9.3a. The corresponding dual nodes are also drawn in the figure, together with dotted lines representing the edges. There are two edges in a straight line, and the only way to implement this with patches is a corner merge as shown in Figure 9.3b. As it is a corner merge, this does not need geometry



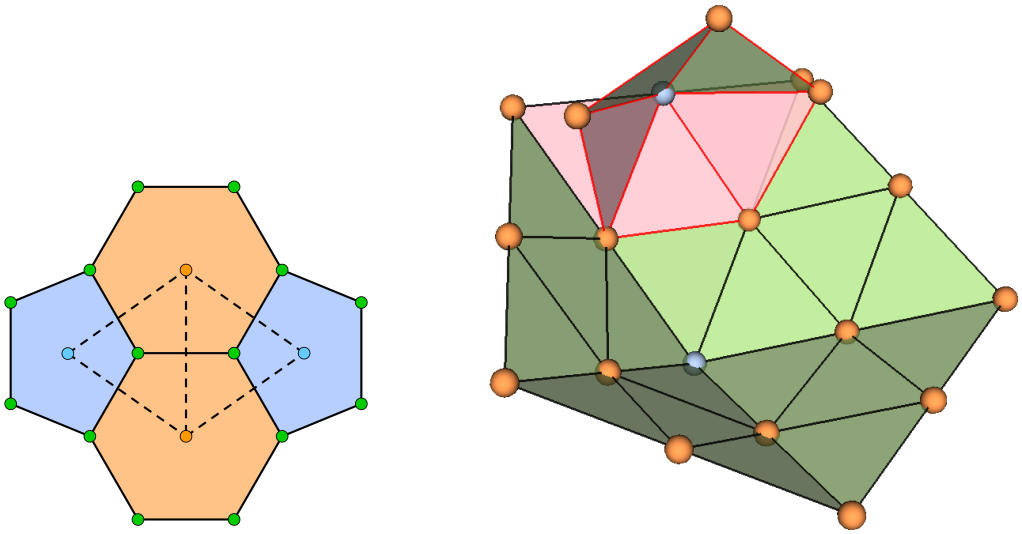
(a) 2 pentagons connected by a shared adjacent hexagon, $d = 2$. This leads to a corner merge as shown in 9.3b.

(b) Corner merge of red $r = 1$ patch into black $r = 2$ pocket region. The geometry is intact, overlap triangles are shown in light red.

Figure 9.3.: A $d = 2$ corner merge, in (a) the constellation of the involved pentagon faces in the cubic representation, in (b) the positioning of the two corresponding dual embeddings.

fixing. This is also visible, as all overlap triangles coincide perfectly and the red patch can flawlessly be merged into the black pocket region.

On the other hand, two pentagons can be separated by two adjacent hexagons, as sketched in figure 9.4a. In that case, one has to merge along a flat area in one sector of the pocket region. Figure 9.4b shows that situation. Again, the overlap triangles are painted in red for both the patch and the pocket region. Not all nodes overlap in this merge: there are of course nodes from the patch that don't have an equivalent in the pocket region. But most importantly, there are two nodes in the patch, which are part of the overlap triangles and their pocket region equivalents have different coordinates. When nodes that



(a) For a sector merge, the cubic pentagons need to have two common triangles.

(b) Sector merge of a $r = 1$ patch (red) into a $r = 2$ pocket region (black). The overlap triangles on both embeddings are coloured in light red.

Figure 9.4.: Similarly to above, here a $r = 2$ sector merge is shown, in (a) the cubic grid formation which is typical for this merge type and in (b) the dual embeddings that are to be merged.

define an overlap face don't overlap for patch and pocket region, then there are always pairs of nodes that have the same global index, that means that a decision has to be made about which coordinates to keep. Here, the case is pretty clear: The red patch is a minimal size patch. In chapter 8.3 when introducing rigid nodes, it was made clear that those can not be moved to adjust the shape of the patch. Meanwhile, the second layer nodes of the black pocket region is not rigid and can be moved.

$D = 3$ and higher merges

For $d = 3$ merges, there are multiple possible scenarios. Working out all the situations goes beyond the scope of this thesis and there were no molecules encountered where this was needed (I generated pocket regions up to a molecule size of C_{120}). However, working out merges with bigger distances between pentagon nodes is definitely a task for the future.

9.2. Geometry Fixing of Pocket Regions

Pocket regions are in general not in a state that allows a merging process, after all patches have been positioned. The $r = 2$ sector merge needs geometry fixing. Because this is the most frequent merge for IPR Fullerenes andrelevant to some non-IPR molecules, it needs to be given some thought. Based on the situation, there are different methods that one can apply to get the desired result.

In general, geometry fixings solve issues that arise when one attempts to merge patches that have been positioned by face overlaps. That can for example be an edge length shorter or longer than unit length, or two nodes that should be overlapping but aren't. The question to ask is "If the given patches were to be merged now, would any inconsistencies in the geometry arise?". If the answer is yes, geometry fixing is needed.

In the following, three methods will be worked out to handle different situations.

The `edgelength` method

A good point to start is a sector merge of two patches with radii $r_1 = 2$ and $r_2 = 1$, which was already used as example for sector merges in Figure 9.4. It is the easiest possible situation that involves a flat merge, and all other situations involve more complicated embeddings or more patches.

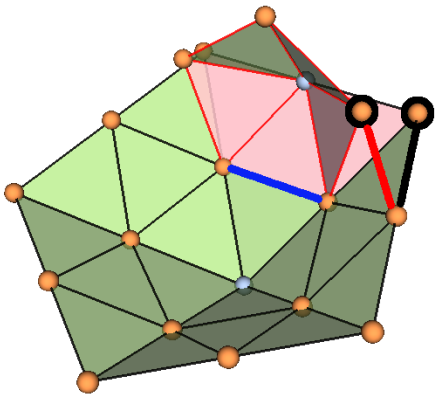


Figure 9.5.: Execution of a sector merge: The black circled node in the black patch is replaced by the black circled node in the red patch. The thick black edge becomes the thick red edge, which is shorter than unit length. The patch needs to be rotated outwards, the thick blue edge is the rotation axis.

Figure 9.5 illustrates what leads to a geometry problem when merging these two patches. The black and red patch both have a black-circled node. This is one of the node pairs from above that do not overlap, yet they have the same global index. The red patch is a minimal size patch and therefore rigid, which means that there is no choice but to select

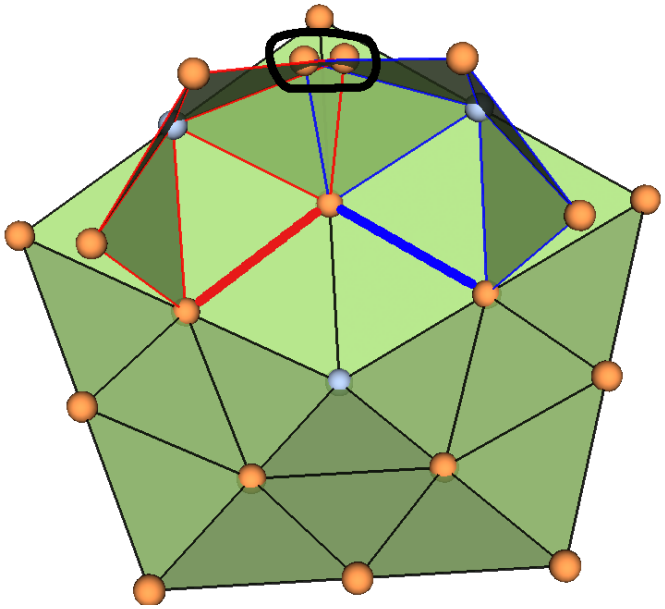


Figure 9.6.: Two patches (red and blue) placed in adjacent sectors of a third patch (black). They have common nodes, which need to have the same position (black circle). To achieve that, they are rotated outwards symmetrically, using the thick red and blue edge as rotation edges.

the coordinates of the red patch's node for the combined embedding. The edge that is indicated as a thick black line in Figure 9.5 becomes the thick red edge, which is shorter than unit edge length. That is what necessitates a geometry fixing. Because this is a symmetric problem, there is also a shorter-than-unit-length edge on the other side of the sector that the merging takes place in. The solution to this problem also has to be symmetric in the same way.

The solution is to rotate the red patch outwards along a rotation axis that is marked blue. Because the whole patch red patch is rotated and the rotation only changes the position of overlap triangles in the black patch, but not their shape, no new geometry problems arise while solving the existing one. As we will see momentarily, rotations can solve all the geometry problems that arise in sector merges.

The `pointdist` and `symm_pointdist` methods

The merge of two grids with radius $r_2 = r_3 = 1$ into a pocket region with radius $r_1 = 2$ produces a situation that requires the simultaneous adjustment of two patch positions. The situation is quite similar to the one in Figure 9.5, only that there are merges in two adjacent sectors of the center patch, it is shown in Figure 9.6.

Again, the geometry problem is caused by two nodes having the same global index, but not the same position. Only that here, both nodes are part of a minimal size patch and can not be moved, which means that deciding to pick one of the two positions would cause another geometry flaw. Figure 9.6 shows an example, where the two nodes are circled in black. Instead of choosing one node position, both patches have to be rotated outwards along their rotation edges, which are marked in thick red and blue respectively. The implementation of this method below will show that there is indeed a rotation angle, at which the node positions coincide.

Now, what if one wants to do another sector merge in an adjacent sector to those two geometry-fixed patches, which have just been rotated such that their common nodes overlap? The red and blue patch from above do not have any degrees of freedom left, as any movement would cause a geometry flaw again. Therefore, a third geometry fixing method can be introduced, combining the comparison of node positions from the `symm_pointdist` method with the single-patch rotation from the `edgelength` method. The condition is the same as for `symm_pointdist`, the absolute distance of two nodes that have identical global indices is minimized. This time though, only the new patch is rotated and the other one serves as comparison. This is for example needed when computing a pocket region for C_{60}–I_h, each of the five sectors is subject to a $d = 2$ sector merge, of which the first two are executed with `symm_pointdist` and the last three with `pointdist`.

Method Summary

In summary, three methods were defined to deal with geometry flaws arising from merging patches:

1. `edgelength`: Optimize single patch position such that edge lengths connecting to pocket region are one.
2. `symm_pointdist`: Optimize 2 adjacent patches symmetrically, such that their overlapping nodes have the same coordinates.
3. `pointdist`: Optimize a single patch, such that a node that overlaps with the pocket region, has the same coordinates as the respective pocket region node.

9.3. Numerical Implementation

We now turn to the more technical part and describe the most important parts of the code that executes all the necessary steps, as well as the concepts that are used for the computations. At this point, I would like to point towards the overview sketch in Figure 9.1 for and overview of what is to come. The more theoretical part above focused a lot on the geometry fixing, but the first two parts of the combination step are the selection of a center patch and the initial placement of the other patches.

Center Selection and Initial Patch Placement

Even though the center selection is a part of the combination step, the decision for the center patch has implicitly already been taken in the patch grouping section 8.1. It was mentioned there that the groups are sorted after how "centered" the pentagon nodes are. In the same way that patches are constructed from the inside outwards, one chooses to construct pocket regions starting at the most "centered" patch, in terms of how central it is in relation to all other patches in the pocket region. Therefore, for each group, the 0'th entry is selected as center patch.

Next, let's take a quick look at the constructor of the pocket region class. It works very similar to the class constructor of the patch class in Listing 8.2, with the difference that it takes the patch object as input that has been selected as center patch and just copies all of its attributes. At this point, the pocket region is identical to the center patch.

```
class PocketRegion():
    def __init__(self, cPatch):
        self.positions = cPatch.positions
        self.triangles = cPatch.triangles
        self.dual_neighbours = cPatch.dual_neighbours
        self.global_index = cPatch.global_index

        self.rigid = np.zeros(self.nodes.shape[0], dtype=bool)
        self.rigid[0:6] = True

        self.patches = [cPatch]
```

Listing 9.1: PocketRegion class constructor.

However, at this point the other patches in the pocket region have not yet been added to the `patches` list. If you remember the class diagram of `PocketRegion` from Table 6.2, there was a method called `addPatches`. Using this method, the other patches are added to the list.

At this point, the initial placement of the first patches in the pocket region is possible. This is where the overlap triangles become useful, so I will give a short summary to connect the dots and refresh the memory. Overlap triangles were mentioned first when drafting a concept for the algorithm, as a tool to combine embeddings. The mechanism for patch radius calculation ensures an overlap between neighbouring patches (see equation (8.1)). Chapter 9.1 showed that for merges with a distance between pentagon nodes of $d = 1$ or $d = 2$, the overlap is always either two or three triangles, and the number of overlap triangles is specifically bound to the type of merge. Corner merges have two overlap triangles, while sector merges have three.

Now, because the construction is done in three dimensions, to determine the position of any rigid object uniquely, the coordinates of three points on the object are needed. Therefore it is ideal to position a patch by specifying the position of a single triangle, which consists of exactly three nodes. If only one triangle is needed, which of the two or three overlap triangles is best chosen? For corner merges, it doesn't matter. Because

there are no flaws in the geometry arising, one can arbitrarily choose one of the two triangles. For sector merges, that is different. It is strictly necessary to choose the central of the three overlap triangles for initial positioning to ensure the symmetry of the merging process. This is quite intuitive when looking at the overlap triangles for example in Figure 9.4b.

When the overlap triangle has been identified, two sets of coordinates are necessary for it: The triangle's coordinates in the pocket region, and the coordinates in the patch. Then a three-step algorithm overlaps the triangles in three dimensions using the `translate()` and `rotate()` functions defined in chapter 7.2:

1. Choose a node from the triangle (node with identical global index, once from patch and once from pocket region) and `translate` the patch such that both nodes have identical coordinates.
2. Calculate the normal vectors of both triangles, then rotate the patch, such that the normal vectors align.
3. Choose a triangle edge that starts at the node from step 1 (both triangles). Then rotate the patch, such that the patch version of the edge aligns with the pocket region version.

Golden Section Search

To overlap the points in *pointdist* and *symm_pointdist* or get to the correct edge length in the *edgelength* method, one could in theory write down equations and solve them numerically. This is unnecessarily complicated though. Each of these problems has one condition and one unknown.

The conditions are to overlap the coordinates of two input nodes, or to optimize an edge length. Because an edge is defined by it's two end points, the conditions look very similar when expressed mathematically. Let $\vec{p_1}$ and $\vec{p_2}$ be the two points to have an identical global index (`symm_pointdist`, `pointdist`) or to define the edge that is too short (`edgelength`). Then they must fulfill the condition

$$\|\vec{p_1} - \vec{p_2}\| = 0 \text{ or} \tag{9.1}$$
$$\|\vec{p_1} - \vec{p_2}\| = 1 \tag{9.2}$$

for a method that optimizes point distance or edge length respectively.

The unknown for any of the methods is the angle that the patch(es) has (have) to be rotated, in order to fulfill the condition. Even when using `symm_pointdist`, there is only one unknown, because the patches always get rotated by the same angle to ensure symmetry.

With a system like this, it is easiest to use one-dimensional optimization methods to find the optimal patch position. I decided to go with the golden section search, because it

is easy to implement and the given problem has a deciding property called *unimodality* that is required for the golden section search [8].

Definition 9.3.1. Let $f : \mathrm{R} \to \mathrm{R}$ be a function. f is called *unimodal* on the interval $[a, b]$, if there is a unique $x^* \in [a, b]$, such that $f(x^*) < f(x) \forall x \in [a, b]$ and for any $x_1, x_2 \in [a, b]$ with $x_1 < x_2$

$$x_2 < x^* \Rightarrow f(x_1) > f(x_2) \text{ and } x_1 > x^* \Rightarrow f(x_1) < f(x_2) \tag{9.3}$$

If the problem is described by a unimodal function, the golden section search can be used to identify the minimum x^*. In the case of the three methods introduced above, there is no function can be mathematically defined, but numerically, f is a function that takes the rotation angle of the patch(es) as an argument and returns the distance of the two nodes.

$$d = f(\theta) \tag{9.4}$$

And while there is no way of proving that this function is unimodal without a mathematical definition, it is quite intuitive to see the unimodality. An example is shown in Figure 9.7, the exact configuration is two $r = 1$ patches being merged into an $r = 2$ pocket region with the `symm_pointdist` method (see Figure 9.6). The red plot shows the absolute distance between the comparison points. The minimum around an angle of 0.2rad is on an absolute distance of 0, just like it should be. The vertical black line indicates the distance at the starting point of the optimization process, after the initial placement of the patches. It is clearly visible that this function is unimodal and therefore golden section search can be applied to find the minimum.

Unimodality can be used in the context of one-dimensional optimization to narrow down the interval which the minimum of a function can be in further and further until the desired accuracy is reached. Two points x_1 and x_2 can be chosen and then equation (9.3) can be used. If $f(x_1) > f(x_2)$, $x_1 < x^*$ is necessarily implied and therefore x^* can not be in the interval $[a, x_1)$. On the other hand, if $f(x_1) < f(x_2)$, $x_2 > x^*$ follows, therefore the interval $(x_2, b]$ can be excluded [8]. The golden section search does not only utilize that, it also does it in the most consistent way possible, by using the *golden ratio* $\tau = {}^{(1+\sqrt{5}))}/_{2}$ to select x_1 and x_2 in the beginning of each iteration [8].

$$x_1 = a + (1 - \tau)(b - a) \text{ and} \tag{9.5}$$

$$x_2 = a + \tau(b - a) \tag{9.6}$$

With these selections, either $[a, x_1)$ or $(x_2, b]$ is excluded from the interval by setting $a = x_1$ or $b = x_2$ and then starting the next iteration.

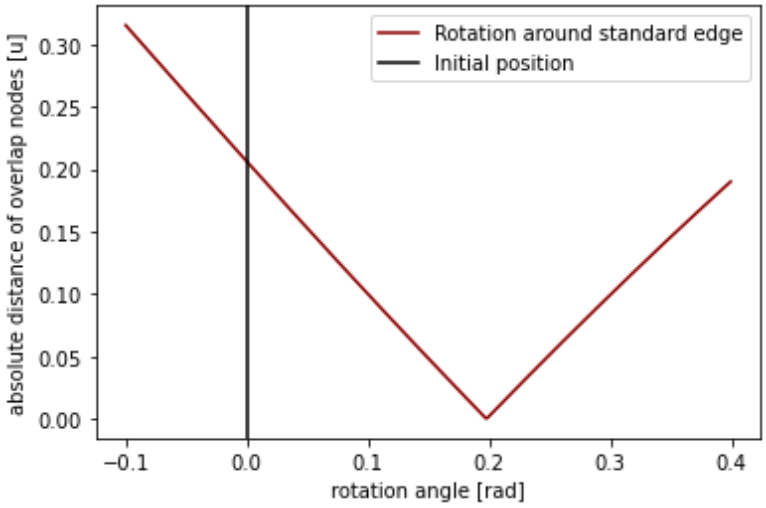


Figure 9.7.: Return values of the `opt` function (i.e. absolute distance of comparison points) for a situation like the one shown in 9.6, where the `symm_pointdist` method is applied.

Algorithm Outline

With the golden section search, all three algorithms can be described in a very similar manner:

1. Input: rotation axis of grid(s), 2 points for condition: Either overlapping points or edge endpoints.
2. Setup of golden section optimization: implement function that rotates grid(s) and returns condition value (distance between points).
3. Find ideal rotation angle around rotation axis with golden section due to condition.
4. Rotate grid(s) by ideal angle. Grid(s) is/are now ready to be merged into the pocket region.

To make rotations possible, there has to be an automatized decision on the rotation edge. As the sector merge always has three overlapping triangles between two grids, the selection of the rotation edge is made according to two rules:

1. The rotation edge has to be part of the central triangle, which was used for initial patch placement.
2. An edge that lies on the boundary of the minimal size patch, has to be selected as rotation edge.

These two conditions make a unique selection of the rotation edge possible.

The Optimization Function

Next, the function f for the golden section search has to be implemented.

```
1  def opt(angle):
2      #Make rotation and extract coordinate data for the edge length.
3      fcs.rotate(patch, rotVec, refP, angle)
4
5      #Compute edge length:
6      re = np.linalg.norm(refPatch.nodes[compareRef] - patch.nodes[
           comparePatch])
7
8      #Revert rotation, such that we are back at the original point
9      fcs.rotate(patch, rotVec, refP, -angle)
10     return re
```

Listing 9.2: *opt* corresponds to the function f that is optimized by the golden sector search. This specific example is from the *pointdist* method.

As it is required (see equation (9.4)), the function takes an angle as argument and returns a distance. The angle input tells the function, how much to rotate the patch. Zero-point of the rotation is always the standard position that the patch is in after the positioning-process. The rotation interval is fixed to $[-\frac{\pi}{2}, \frac{\pi}{2}]$. This interval covers half of all the possible rotations. Given that the initial positioning is quite close to the fixed geometry and only rather small corrections are needed, the solution will always be covered in this interval.

The `opt` function is defined only for the golden section search and therefore inside of either of the three geometry fixing methods. First of all, there are two patches involved, they are called `patch` and `refPatch`. `patch` is always the patch being rotated, and as the name `refPatch` indicates, that is the reference patch for the distance computation. In the `edgelength` example in Figure 9.5, the reference patch is the patch with black edges. In the `opt` function for the `symm_pointdist` method, there is no reference patch. Instead, there are two patches, both being rotated by the same angle.

There are also two variables called `compareRef` and `comparePatch`. These hold the indices of the nodes that are used for the distance computation. The indices are defined in the index layers of the `patch` and `refPatch` objects. Last but not least, there are the inputs of the `rotate` functions. For a summary of how the `rotate` functions work and what the purpose of all the inputs is, I refer to chapter 7.2. I do want to point out though, that the `rotVec` parameter corresponds to the rotation edge of `patch`.

So what does the `opt` function actually do? The procedure consists of three main steps:

- Line 3: *patch* is rotated around its rotation edge, from the zero-point that is defined by patch positioning and by an angle in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$.
- Line 6: The distance between two specified nodes is computed. This is what the function returns as output.

- Line 9: *patch* is rotated back to the zero-point position

The `opt` function in the `edgelength` method has a small addition in the second step, after line 6 in the listing above.

```
1 re = np.linalg.norm(re - 1)
```

The return parameter is not the pure distance, instead its difference to 1. The reason for that is that the target distance in the edgelength method is $d = 1$ instead of $d = 0$, as it is in the other methods. This way, it is made sure that `opt` is a unimodal function.

With this function, the golden section search can be executed to find the optimal rotation angle for a patch. When this angle has been found, a final rotation can be made to bring the patch into optimal position, and then the geometry is fixed.

9.4. Rigid Nodes and Remnant Geometry Flaws

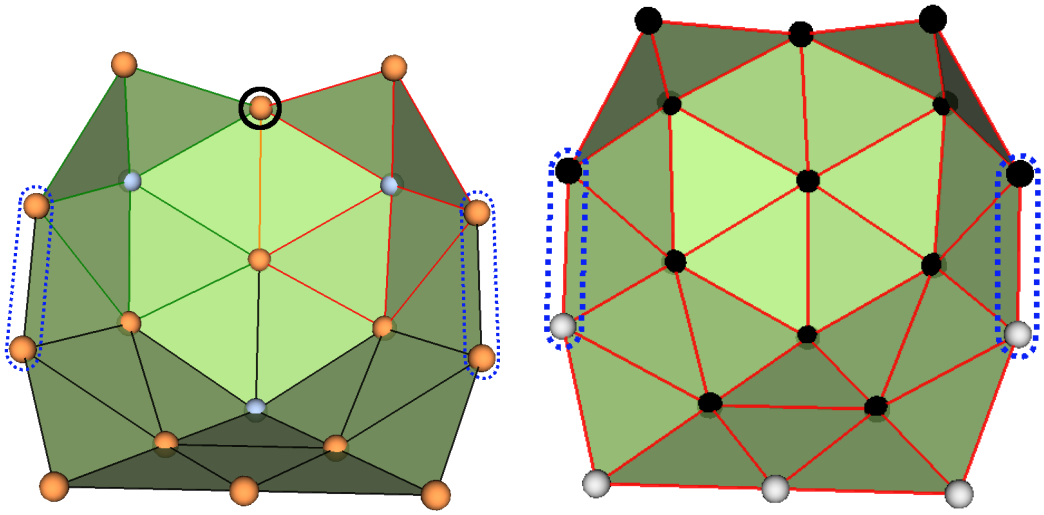
In certain situations, inconsistencies in the geometry of the pocket region embedding can arise that are not solved in a geometry fixing. The most common of these situations is two adjacent flat merges of $r = 1$ patches on a $r = 2$ pocket region. This is a recurring example, shown again in Figure 9.8.

This configuration is the prime example for using `symm_pointdist` as geometry fixing method. Theoretically, it is also possible to use the `edgelength` method for fixing both patches separately. The optimization conditions for these two paths are independent from each other. The edges circled in dotted blue in 9.8a would be used for `edgelength`, If the geometry of the pocket turns out to be consistent after the merging process, these approaches would have equivalent results. However, this is not the case. This causes a remnant geometry flaw, two edges in the embedding are to short.

The situation, this time with the already merged pocket region, is shown in Figure 9.8. The edges that would have been optimized by the *edgelength* method, are marked in thick blue in both the left and the right pocket region. When merging with the `symm_pointdist` method, these edges that would be used for the `edgelength` method optimization, are too short. The other way around, the common node of the green and red patch that `symm_pointdist` uses to optimize the geometry (that node is marked with a black circle in 9.8a) are not overlapping correctly. The methods are not fixing the geometry consistently.

9.8b shows the rigid and non-rigid nodes in this pocket region. Rigid are right now all nodes that are in a minimal size patch, these are rigid from the moment the patches are generated and the `rigid` property is copied from patches to pocket regions when merging. The node fixed by `symm_pointdist` already has the rigid property. Therefore it is crucial that this geometry flaw is fixed before the merging, i.e. in the geometry fixing step.

On the other hand, the shortened edges which would need to be fixed by the `edgelength` method connect a rigid and a non-rigid node. This means, their length can still change



(a) The black circled node was used for the `symm_pointdist` method, leaving the shortened edges circled in blue.

(b) Rigid (black) and non-rigid (light grey) nodes marked.

Figure 9.8.: $C_{120}-T$ pocket region after geometry fixing and merging. The geometry fixing with `symm_pointdist` leaves two shortened edges circled in dotted blue lines (a). This is not a problematic situation though, because the nodes in the lower ends of those edges are not rigid yet (b).

by changing the position of the non-rigid node, even after the merging. This happens in the fill-up algorithm, when layers of the pocket region are completed, which means that the positions of all nodes in a layer are unanimously determined and their status can therefore be set to rigid. Because all edge lengths to existing rigid nodes are checked and corrected in that process, the fill-up algorithm automatically takes care of the geometry flaws.

9.5. Patch Merging Process

Only now, after all patches are in position, comes the actual merging operation. This part of the algorithm fully includes the new patch(es) into the pocket region, by adding the positions of all new nodes, changing the positions of existing nodes if necessary and including new triangles. All of this is done in a single method of the `pocketRegion` object, which is called `mergeFaces` and relies on identifying patch and pocket region nodes through the global index layer, in order to add all new nodes from the patch to the pocket region, as well as the new triangles. Another important transfer is the information on which nodes are rigid and which aren't.

The merging process is quite a tedious and organizational task that is necessary, but going

through it does not yield great benefits in the understanding the algorithm. Therefore, the code is not addressed in this section and instead explained in appendix A.

10. Growing Step

Now the combination step is finished and all pocket regions have their main shape-building features set up. For some pocket regions, this might be the end result already, if there is an intact layer structure and no need for growth. For most pocket regions, at least the layer structure needs to be set up.

What Does it Mean to "Grow" a Pocket Region

"Growing" a pocket region in this algorithm means that more hexagon nodes are added around the central embedding of the pocket region. Still, it can only be hexagon nodes that are added in layers, such that the shape influence of all pentagon nodes is respected. If the embedding has to be so big that additional pentagon nodes would have to be added when growing from a single pocket region, then merging multiple pocket regions instead is the better way to go. The reason is once again that the pentagon nodes are shape-defining in areas close around them and building into a pentagon node does not respect that concept.

But growing pocket regions is best done in a structured way. Chapter 4 discussed the necessity to give pocket regions a layered structure. This layered structure can then be continued by adding full layers of hexagon nodes. A layered structure depends greatly on selecting a center to start from. For patches, this was trivial: the pentagon node is the obvious center of each patch. For pocket regions, the decision is not quite as easy. Selecting a center for the layer structure will be the first point addressed.

After selecting a center, a pretty clear path emerges: Continue the layered structure. With the help of the `nextLayer` function, the next layer can always be computed in the bond graph and in the pocket region. Then, both layers can be compared. If there are nodes and triangles missing in the pocket region, they can be added. This is done, until the pocket region has reached the desired radius.

This process is split up in two parts: The "Fill-Up" part refers to the construction of layers that already exist in part. That is, in a pocket region that has one or more outer layers which are not finished, nodes are added until the layer(s) is/are finished.

The fill-up part always has to be completed, such that the pocket region has a layered structure in the end. When this is done though, the "Growing", which adds entirely new layers, is optional.

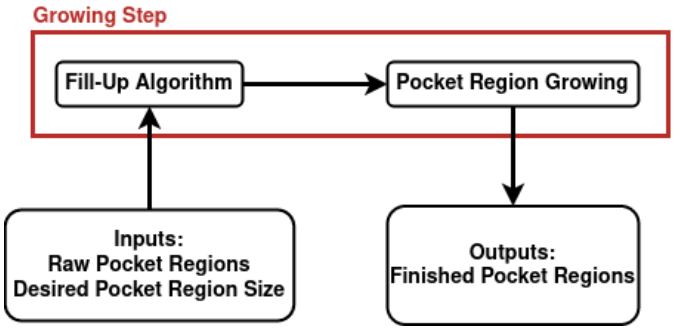


Figure 10.1.: Overview of the last major step of the algorithm, the growing step.

About The Purpose of This Chapter

This chapter reaches the point, where in many stages, there does not exist an implementation for what is described. In some cases, there is also not a concept for how to solve problems. For that reason, this chapter will mostly discuss the approaches that could be taken or are in development. Where it is possible, there will also be a discussion of implementations.

10.1. Center Selection for Pocket Regions

Even though, one can not simply select one of the pentagon nodes to be the center of a pocket region, it is still the deciding factor of selecting a center to look at how the pentagon nodes are arranged. In chapter 4.3, the example of a three-patch pocket region was mentioned, with a highly symmetric node in the middle of all three pentagon nodes that can be chosen as midpoint. It is quite easy to write a function that finds this node: One can select the node that has the least average distance to all pentagon nodes. This was also used to select the center patch in the combination step, only that the selection was restricted to pentagon nodes back then, whereas now any node can be selected.

However, there are cases where no node can be selected that is the symmetric center of a pentagon node constellation. In Figure 10.2, several constellations of pentagon nodes are shown. All of these can not have a center that is on a single node. The four-node constellation can either have the two inner triangles or an imaginary point in the middle of the four nodes as center. The best choice for centers in the middle figure is the connecting edge, and in the right constellation it is the central triangle. Currently, there is no concept that clearly determines a center for each possible constellation. However, for the pocket regions that have a node as center, as in the case of $C_{120}-T$ from chapter 4.3, the fill-up algorithm can be started. We will come back to that example later.

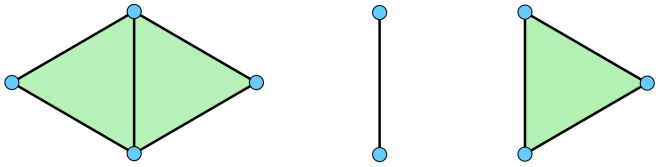


Figure 10.2.: Three examples for pentagon node constellations in Non-IPR pocket regions. The centers of these constellations are the two triangles (left), the edge between the nodes (middle) and the single triangle (right).

10.2. Node DOF's

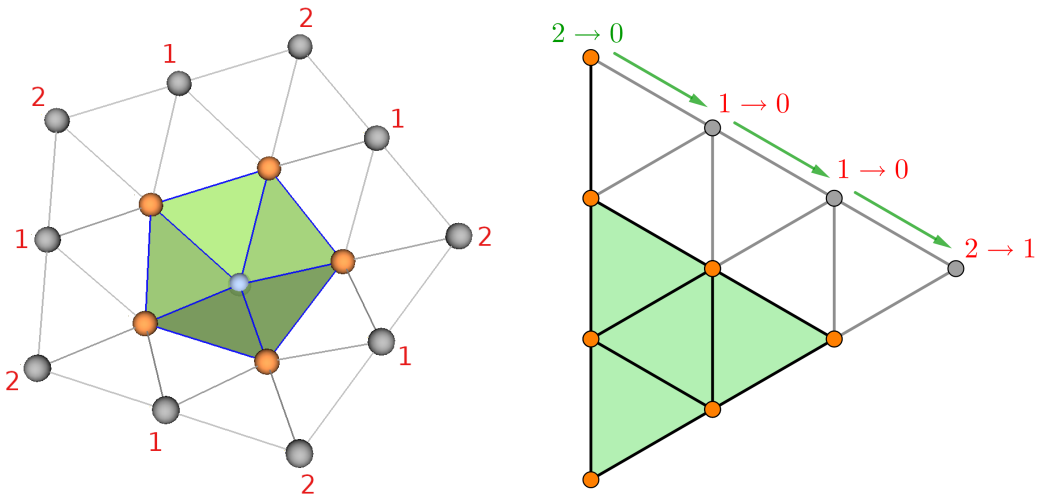
Growing a pocket region means to determine the positions of new nodes, such that all restrictions from the geometry are satisfied. A node's position corresponds to three unknowns, because the embedding is in three dimensions. In other words, the node's position has three degrees of freedom (DOF). There are also constraints on a node's position; connections to other nodes, edges with a fixed length. If a node has position $\vec{r}$ and an edge leads to another node of position $\vec{c}$, the edge between the nodes introduces an equation of the form

$$\|\boldsymbol{r} - \boldsymbol{c}\| = 1. \tag{10.1}$$

Because only the positions of rigid nodes are unchangeable, fixing the position of any node requires three edges that end at rigid nodes, then the system of three equations can be solved for the three unknown coordinates. Solving this system of three equations of the form (10.1) for a single node position will be addressed in detail in section 10.4.

Single nodes with no degrees of freedom are solvable without too many problems. But in general, one can not expect a node to be connected to three fixed nodes and have zero degrees of freedom. Instead, one will have to assume that a whole new layer is added to a patch or pocket regions, where all existing nodes are rigid. What amount of DOF do the new nodes have in that scenario?

The easiest example that can illustrate this is a simple patch with radius $r = 2$. Figure 10.3a shows what amount of DOF adding a second layer yields for each patch. Now of course the patches are generated in standard shape and the DOF are not important, but this still yields a useful insight: Corner nodes always have two degrees of freedom when being added in a new layer. Sector nodes always have one degree of freedom. This is absolutely logical, because all DOF-reducing equations come from the previous layer, where all nodes are assumed to be rigid. As corner nodes are defined to be connected to the previous layer by one edge, their total DOF are $3 - 1 = 2$. Sector nodes' DOF can be calculated similarly with two connections to the previous layer. Now, there are also edges between the freshly generated nodes, which means that they are not independent from each other. Fixing a single node reduces the degrees of freedom of the adjacent nodes. In fact, it is very beneficial to fix corner nodes with whatever method is possible. Figure 10.3b shows an example in a single sector of a 3-layer patch. The left corner



(a) DOF (red) for all nodes in a new layer. The corner nodes have always 2 DOF, the sector nodes 1.

(b) One corner node is fixed in position, its DOF being reduced from 2 to 0 (green), this cascades along the sector.

Figure 10.3.: Two examples from patch building. In (a), the second layer of a patch is added, in (b) a corner node is fixed in position, the reduced DOF cascade through the adjacent sectors.

node is fixed, the DOF are reduced from two to zero (indicated in green). This gives the adjacent sector node one additional rigid neighbour, so its DOF are reduced by one to zero, it is rigid too now. This cascades through the whole sector, until the next corner node stops it. It is crucial that this is not one big system of equations being solved, it is a series of single-node problems being solved.

All of this yields one big takeaway: The corner nodes are the key to solving the problem. If the DOF of corner nodes can be reduced to zero, the sector nodes in adjacent sectors will follow. In fact, the amount of corner nodes in a layer seems to be identical with the excess amount of DOF in the system of equations in this case, as the new greyed out layer in Figure 10.3a consists of ten nodes, which equates to 30 DOF in total. These are reduced by the edges interconnecting the layer nodes, which is of course the same amount as the nodes themselves, 10. And then there are two additional equations per sector node ($2 \cdot 5$ in total) and one additional equation for each corner node ($1 \cdot 5$ in total), therefore we end up with $30 - 10 - 10 - 5 = 5$, which is exactly equal to the amount of corner nodes. Let's keep this in mind throughout the next section, after which it makes sense to define a more general rule.

10.3. Sectors and Corners in Pocket Regions

In chapter 4.3 it was stressed that the same structure can be introduced to pocket regions as to patches, i.e. layers as well as corners and sectors. However, while layers have seen quite some attention in pocket regions due to the cone structure that is required in a pocket region build, sectors and corners have not been addressed beyond the $C_{120}-T$ example from the very same chapter. This will be done now.

In Figure 4.8, it was shown that a pocket region with three pentagon nodes only has three corners. It is therefore beneficial to take a look at a few more situations to see, if there is a system behind the amount of corners emerging from a pocket region. Figure 10.4 shows four example pocket regions, where 10.4a is the known $C_{120}-T$ constellation. The other three pocket regions yield two, five and six pentagon nodes and have varying amounts of corner nodes and corners. These and other examples seem to follow the following rule:

Proposition 10.3.1. *Let* C_l *be the amount of corner nodes in a pocket region layer* l *and* P *the amount of pentagon nodes enclosed by* l*. Then the following equation holds:*

$$\mathsf{C}_l = 6 - \mathsf{P} \tag{10.2}$$

There is of course no theoretical proof for this statement yet, and it might have to be investigated more thoroughly in the future.

Excess DOF in Pocket Regions

We can now come back to the amount of excess DOF not accounted for by equations in a layer. This was discussed for the example of a two-layer patch in the end of the last section. Now, a more general case for pocket regions can be investigated. Let l be a layer consisting of N_l nodes, of which C_l are corner nodes. Let K be the excess amount of DOF in a layer. This quantity is defined by the difference between the summed up DOF for all nodes and the amount of equations E:

$$\mathsf{K} = 3\mathsf{N}_l - \mathsf{E} \tag{10.3}$$

How big is the excess amount of DOF? One can compute E by writing

$$\mathsf{E} = \mathsf{N}_l + 2(\mathsf{N}_l - \mathsf{C}_l) + \mathsf{C}_l \tag{10.4}$$

Hereby, the first term on the RHS are the edges that interconnect the layer nodes, the second term are the edges going from the old layer to sector nodes in the new layer and the third term are the edges going from the old layer to corner nodes in the new layer. With this, equation (10.3) becomes

$$\mathsf{K} = 3\mathsf{N}_l - [\mathsf{N}_l + 2(\mathsf{N}_l - C_l) + C_l] = \mathsf{C}_l \tag{10.5}$$

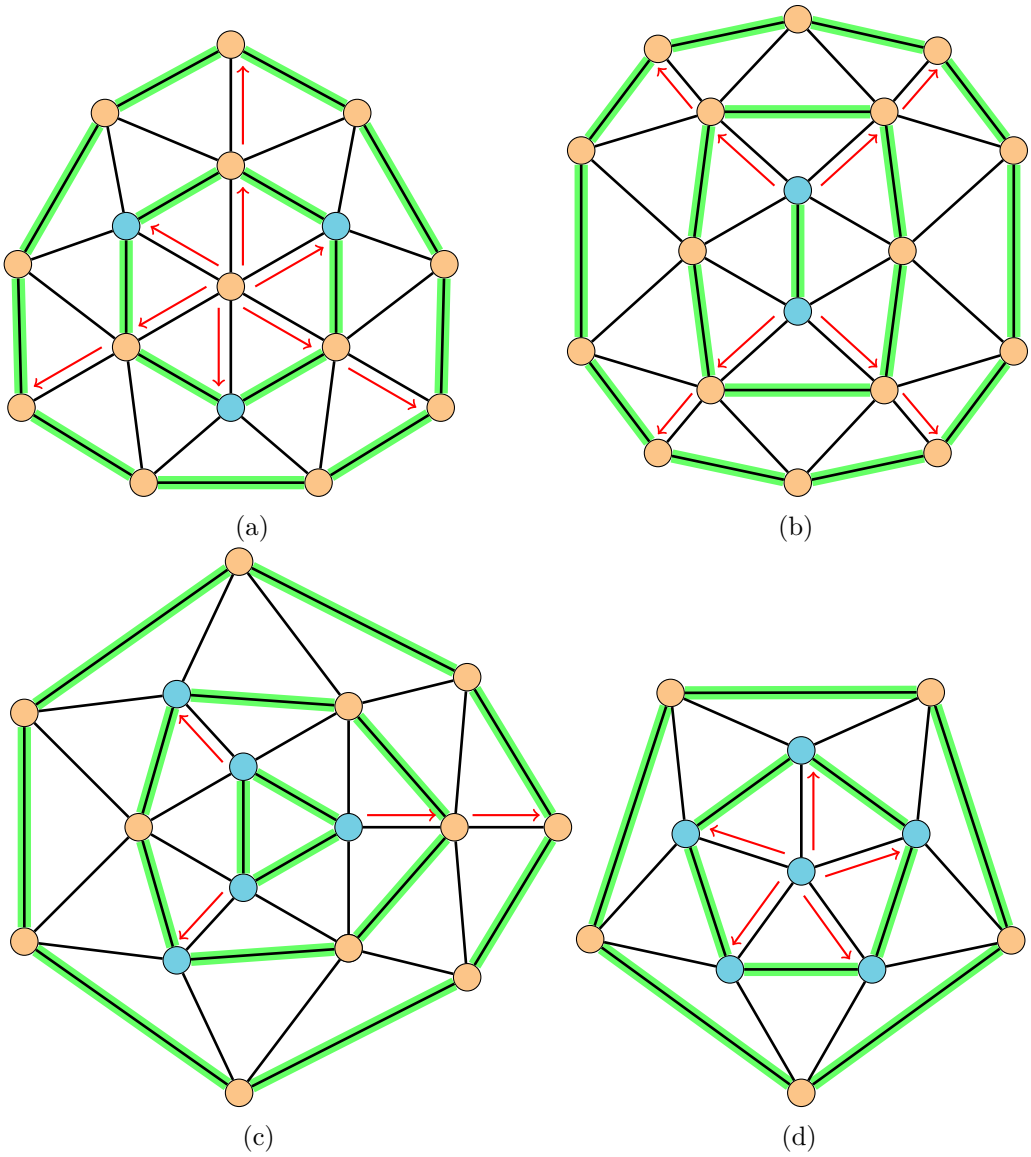


Figure 10.4.: Four examples of pocket regions, layers marked green and the flow of corner nodes marked with red arrows. The examples yield 3 (a), 2 (b), 5 (c) and 6 (d) pentagon nodes.

Therefore, it is indeed the case that the amount of excess DOF in a layer is determined by the amount of corner nodes, and therefore, through Proposition 10.3.1 directly by the amount of pentagon nodes enclosed in a layer.

Reducing Corner Node DOF's

The use that one can draw from these results is that, if each corner node's DOF can be reduced by one, the amount of equations is equal to the amount of unknowns, and each node exactly has one DOF. This system is then definitely solvable, one approach could be to start with one of the nodes and fix its last DOF. Then this fixing cascades around the whole layer, because the cascade is not stopped by the corner nodes anymore.

10.4. The Single Node Solver

A central piece of the growing step is the *single node solver* (SNS). This piece of code can be applied to nodes which have three rigid neighbours and therefore zero degrees of freedom. Their positions can be computed from the distance equations that arise from the edges.

The SNS is not my work, my colleague Nikolai, who is working on curvature smoothing and mesh refinements in Fullerene surface manifold embeddings, has written almost all of it, I have changed a few things to fit it to the needs of my program [11]. In the following, I will explain how the solver works theoretically.

When writing about degrees of freedom in this thesis, we refer to the effective degrees of freedom of a node. Each node has 3 DOF initially, and they get reduced by one for each connection to a rigid neighbour node. Let the positions of the rigid neighbour nodes be $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$; with $p_i = (x_i, y_i, z_i)$. When there are three connections, the three equations

$$(x' - x_i)^2 + (y' - y_i)^2 + (z' - z_i)^2 = l_i^2 \text{ for } i \in 0, 1, 2 \tag{10.6}$$

have to be solved. We will solve the equations for general edge lengths l_i here, albeit the edge lengths in my model are all 1. Shifting the coordinate system $(x', y', z') \to (x, y, z)$ with $x = x' - x_0$, $y = y' - y_0$ and $z = z' - z_0$, the first equation reduces to

$$x^2 + y^2 + z^2 = l_0^2, \tag{10.7}$$

for the other two equations, the coordinate change is absorbed in the positions of the points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. With the help of equation (10.7), the two other equations in the system can be linearized. In matrix form, this yields

$$\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \tag{10.8}$$

The two constants on the right hand side of this equation are $c_1 = \frac{1}{2}(\boldsymbol{p}_1^2 + l_0^2 - l_1^2)$ and $c_2 = \frac{1}{2}(\boldsymbol{p}_2^2 + l_0^2 - l_1^2)$. After a few manipulations, the system of equations in (10.8) yields

$$\begin{pmatrix} 1 & 0 & z_1' \\ 0 & 1 & z_2' \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} c_1' \\ c_2' \end{pmatrix} \tag{10.9}$$

The new constants in this equation are:

$$c_2' = \frac{x_1 c_2 - x_2 c_1}{x_1 y_2 - y_1 x_2} \text{ and } z_2' = \frac{x_1 z_2 - z_1 x_2}{x_1 y_2 - y_1 x_2}, \tag{10.10}$$

$$c_1' = \frac{c_1 - y_1 c_2'}{x_1} \text{ and } z_1' = \frac{z_1 - y_1 z_2'}{x_1} \tag{10.11}$$

$$\tag{10.12}$$

The equations for x and y are now decoupled and therefore these two equations can be used to make equation (10.7) a quadratic equation of z:

$$0 = \left[c_1' + c_2' - l_0^2\right] - 2\left[c_1' z_1' + c_2' z_2'\right] z + \left[1 + (z_1')^2 + (z_1')^2\right] z^2 = \alpha + \beta z + \gamma z^2 \tag{10.13}$$

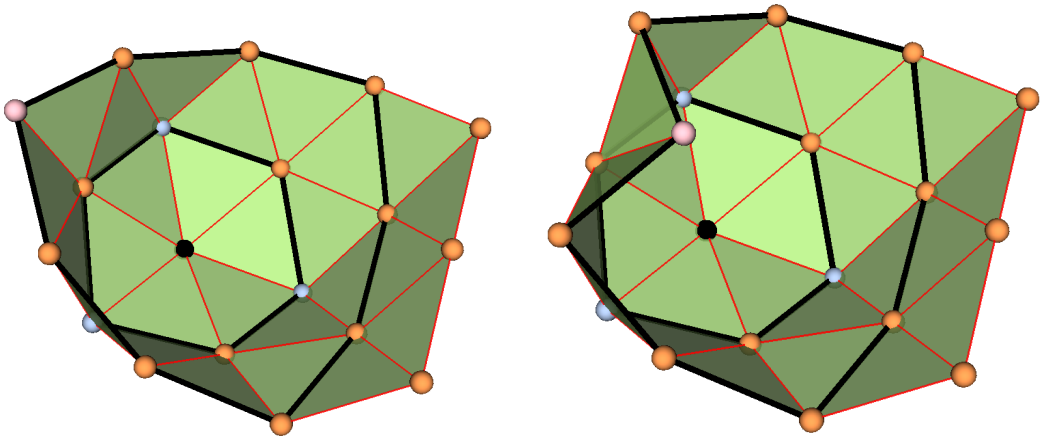
The solutions for z are now calculated by

$$z_{1/2} = \frac{-\beta \pm \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha} \tag{10.14}$$

The last step is now to select the fitting solution of the two. When looking at both of them visually, this is usually an easy choice, because one o the solutions will cause an inwards dent in the embedding, while the other one is nicely directed outwards. Computationally, one can compare the x- and y- coordinates of the solutions and, because every layer lies approximately in a plane parallel to the x-y-plane, the coordinate with bigger $\|(x, y)\|$ is selected. An example for this is shown in Figure 10.5, which picks up the C_{120}−T example pocket region from chapter 5.2 again. The pink node's position is computed by the SNS when applying the fill-up algorithm, which completes the layers that are marked in black. The two solutions that the SNS finds are shown in 10.5a and 10.5b. It is pretty intuitive to find the correct solution, and mathematically it can be perfectly done by choosing the solution with the bigger $\|(x, y)\|$-value.

Additional Conditions

For this method to work in all situations, two safety measures are taken. First, the positions $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ can not be colinear in the coordinate system (x, y, z). In the applications of this thesis, this should never be the case anyways, because the three edges build equilateral triangles between them. The second, more important precaution is to check for linear dependency of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ in the $x - y - plane$.This corresponds to the



(a) Correct position of the light red node (b) Wrong position of the light red node

Figure 10.5.: Pocket Region From the $C_{120}-T$ molecule after fill-up algorithm. The light red node's position has been determined by the SNS from the three connected neighbour nodes.

denominator of the constants in equation (10.10), $x_1y_2 - y_1x_2$ being zero. In that case, the coordinates can be permuted, i.e. $(x, y, z) \to (z, x, y)$ and the algorithm tries again with the new coordinates.

I will not show any code from the single node solver algorithm here, as it is essentially just computing all the coefficients and then solving for z, and with the solution for z computing x and y. The original function is called `calc_vertex_positions` can be found in the file functions.py (see chapter 1.3).

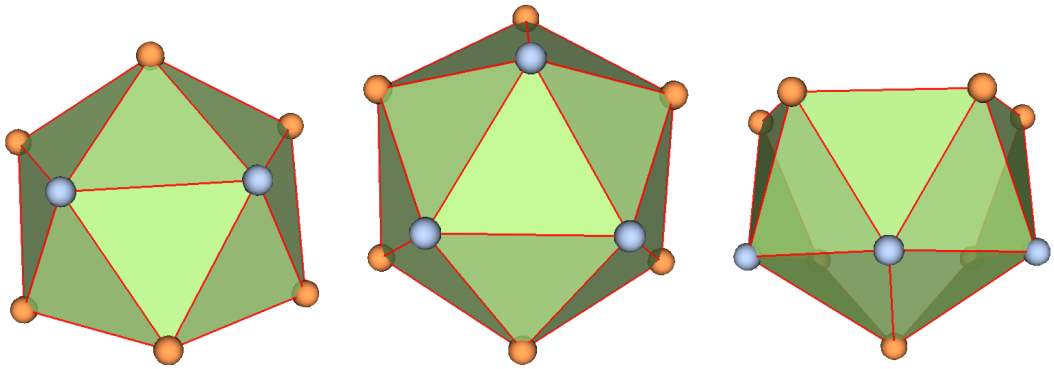
11. Results and Discussion

The beginning of this chapter marks the finish line for the description of all concepts and implementations that were worked out in the course of this thesis. Although in some way, the algorithm itself is the result of the thesis, it is interesting to discuss its capabilities. This chapter will, by going through a few example constructions, try to give a small overview of what types of pocket regions can be constructed by the current implementation and where problems arise. Thereby, the focus lies mostly on the construction and combination step, as the growing step is merely more than pieces of a concept and an initial implementation yielding the single node solver as only tool. The discussion will be split between IPR and non-IPR pocket regions.

11.1. Non-IPR Pocket Regions

One to Three Pentagon Nodes

For non-IPR pocket regions, the only type of merges is the $d = 1$ corner merges. These are quite trivial in the sense that there is no geometry fixing that can go wrong, and up to a pocket region size of three pentagon nodes, they work flawlessly. Figure 11.1 shows all three possible examples for two and three pentagon nodes. These constructions are



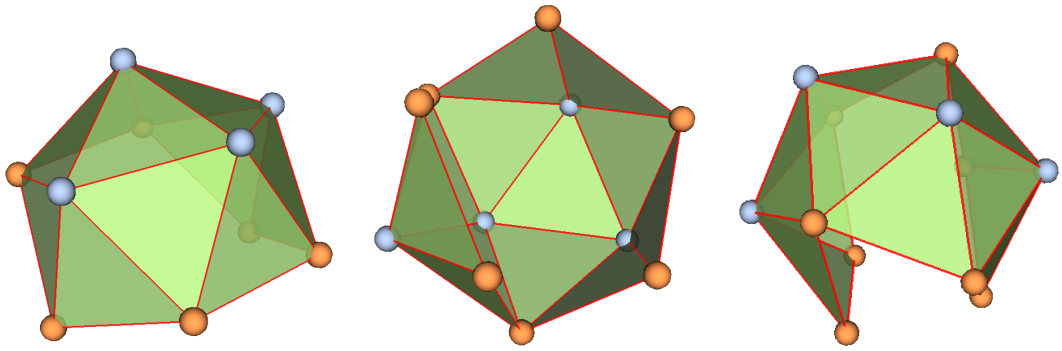
(a) (b) (c)

Figure 11.1.: Possible non-IPR pocket regions with less than four pentagon nodes. With two nodes, there is only one option (a), while three pentagon nodes can build a triangle (b) or a line (c).

quite straight forward, but a good indicator for several processes working well. A quick check shows that all triangles are equilateral and the curvature in the pentagon nodes of all is $\frac{\pi}{3}$ as intended. This indicates that the standard patch construction works well and second, moving and merging patches does not impact the geometry and works as intended as well. The index structure is intact after merging and the triangles are saved correctly. Overall, these smaller pocket regions seem to be constructed correctly with the method.

Four to Six Pentagon Nodes

Having the choice of four to six pentagon nodes, more options for arranging the pentagon nodes open up. Figure 11.2 shows three examples of four-pentagon pocket regions. Construction runs smooth on all of these as well. There are also some pocket regions



(a) (b) (c)

Figure 11.2.: Three selected non-IPR pocket regions made from four minimal size patches. The pentagon nodes form a rhombus (a), a triangle with one outlier (b) and a zig-zag line (c).

that the first implementation of the fill-up algorithm with the single node solver can be used on. This situation arises, when there are single nodes in a layer that have three rigid neighbours. Figure 11.3 shows two example pocket regions First of all, the geometry of these two embedding is correct, all triangles are equilateral and deficit angles sum up to $2\pi - \kappa$. Yet, it is visible that the shapes are not as smooth as one would expect from an optimized model, a result of the rigid edge length constraints. These embeddings are still perfectly valid w.r.t. the geometry constraints. However, because there exists no implementation for a growing algorithm, it couldn't be tested if these shapes can be grown without problems.

The simplest pocket region from six pentagon nodes is the most symmetric one, which is for example a part of non-IPR nanotube-like Fullerenes. Figure 11.4 shows the corresponding embedding, which can also be constructed without geometry flaws. What most of the examples that have been shown so far from four pentagon nodes and upwards have in

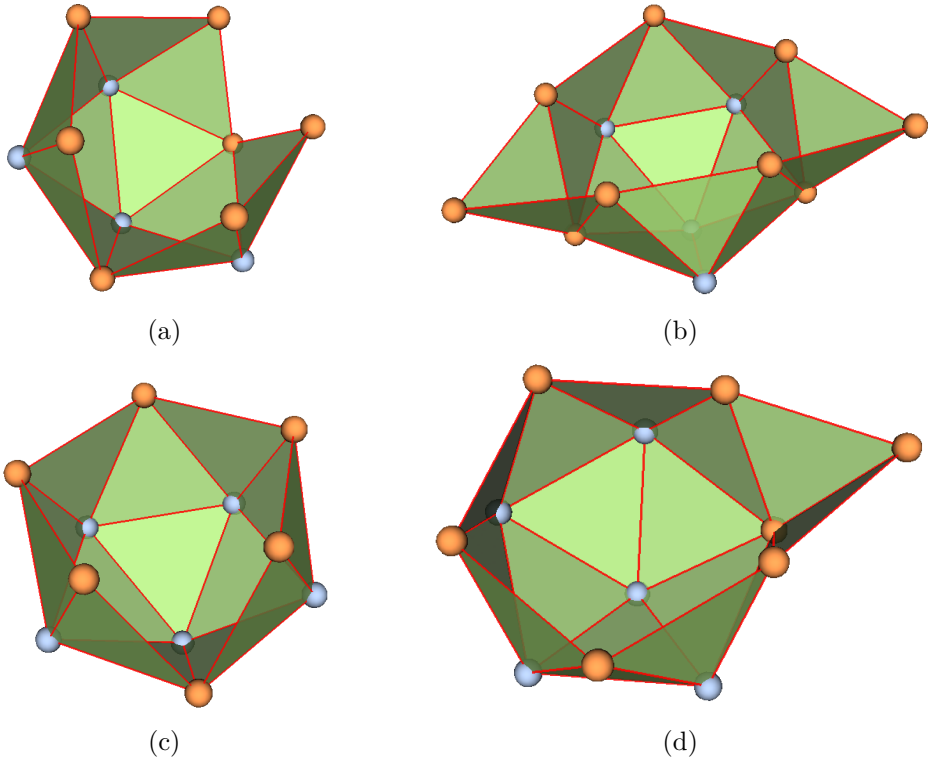


Figure 11.3.: (a) shows the zig-zag four-pentagon pocket region from 11.2b and (c) a maximally symmetric five-pentagon pocket region. To both, the fill-up algorithm can be applied, which yields (b) and (d).

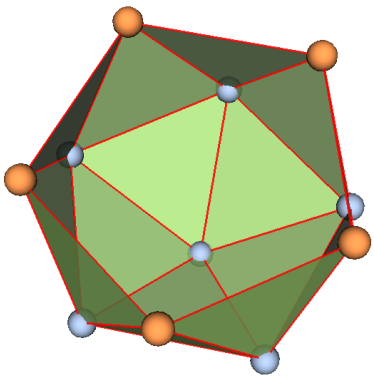


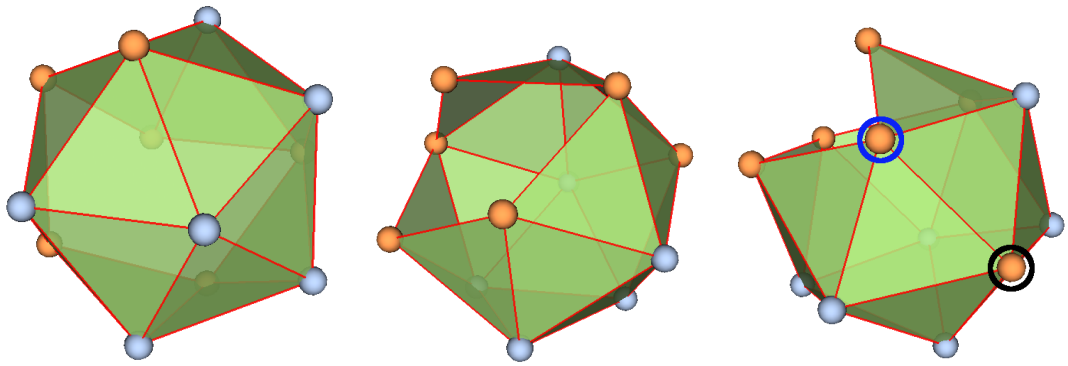
Figure 11.4.: Maximally symmetric 6-pentagon pocket region

common, is that the arrangement of pentagons is very compact. This is most apparent in Figure 11.4 for six pentagon nodes, whereas the zig-zag line of four pentagon nodes from 11.2c is the exception. If the pentagon nodes are arranged in the pocket region in a less compact way, it is more likely that the construction of a geometry flaw-free embedding does not succeed. There are examples for this in four-, five- and six-pentagon pocket regions.

Cases of Flawed Construction

Pocket Regions that feature four to six pentagon nodes can have serious problems with shape. Here an example with six pentagon nodes is shown where the arrangement of pentagon nodes is changed from the maximally symmetric case (see figure 11.4) to a less symmetric arrangement. There are a few good examples in C_{60} nanotube-like Fullerenes, because those are all Non-IPR molecules.

Figure 11.5 shows the pocket region from three different angles. In 11.5a, the arrangement of pentagon nodes is in the focus. Instead of having a center pentagon node surrounded by five more pentagon nodes, as regular nanotubes have, one of the nodes surrounding the nanotubes is a hexagon node and the last pentagon node is further out. This leads



(a) (b) (c)

Figure 11.5.: Comparison of this special form of a 6 pentagon pocket region to the standard version.

to a smaller opening of this pocket region, as 11.5b shows. In fact, this looks like the Gaussian curvature carried in the pocket region is higher than 2π, because there is more curvature than in a half-sphere. The reason for why it looks like this is the focus of 11.5c. The hexagon node circled in black is the node that was also the focus of 11.5a; it replaces the sixth pentagon node in what would be a regular, symmetric 6-pentagon pocket region. This node has five triangles around it in a shape that looks exactly like a minimal size patch around a pentagon node. There is one face missing here. In fact, the face is not constructed yet, because it is not part of any minimal size patch. But it is not possible to construct it properly, because the two nodes that are needed to span it

up, coincide in their positions. This is what is shown as a single hexagon node in a blue circle in 11.5c.

This poses a serious problem. The pocket region does not have any obvious degrees of freedom, by which the geometry can be changed easily, and construction can not be continued with the fill-up algorithm, because the triangle that is supposed to be spanned between the node in the black circle and the coinciding nodes in the blue circle, can not be equilateral. At the same time, there are no violations of the geometry rules in other parts of this embeddings. All triangles and deficit angles are correct.

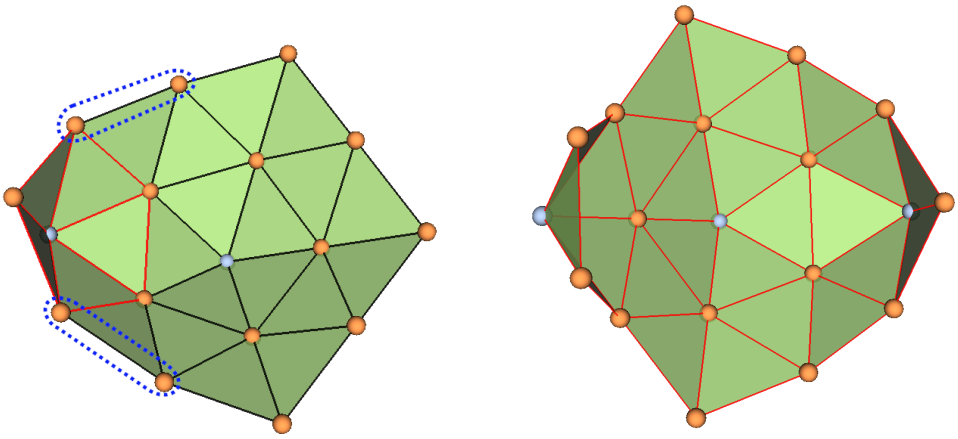
Alexandrov's uniqueness theorem guarantees that there must be an embedding for this pocket region though, so a solution must exist. One of the next steps to investigate this could be to take another look at the minimal size patches. So far, changing their shape was avoided, as it was not necessary and too complex for a quick construction. However, there are some degrees of freedom in a minimal size patch, so that could be the next approach.

11.2. IPR Pocket Regions

Pocket Regions with isolated pentagons are of great interest, because they make up the majority of pocket region in bigger Fullerenes. As this thesis started with small and medium-sized Fullerenes up to C_{120} isomers, not a huge selection of pocket regions was actually generated. Still, they were investigated thoroughly, mainly the ones that needed sector merges and geometry fixing.

One to Three Pentagon Nodes

Similarly to non-IPR pocket regions with this amount of pentagon nodes, there are no construction problems with small pocket regions. Pocket regions with under four pentagon nodes can not make use of the `pointdist` method (one needs at least one center patch and two patches positioned with `symm_pointdist` before that makes sense), the other method to look at is `edgelength`. Two examples which the `edgelength` has been used on are shown in Figure 11.6. 11.6a shows a simple sector merge of two patches, which an `edgelength` geometry fixing. Afterwards, the pocket region has a flawless geometry. The pocket region in 11.6b is very similar to that, only that one additional minimal size patch is merged on the left side by corner merge. In the course of this thesis, a three-pentagon IPR pocket region from a $C_{120}-T$ Fullerene has been used for several examples. This pocket region is constructed using the `symm_pointdist` geometry fixing method, which works out flawlessly, and can also be used to test the SNS/fill-up algorithm. Both cases are drawn once again in Figure 11.7. This case has been addressed extensively, for example in the course of the fill-up algorithm SNS discussion in section 10.4 or in the chapter about geometry fixing and remnant geometry flaws in section 9.4, where it was discussed that this pocket region does actually not have a flawless geometry, as two edges



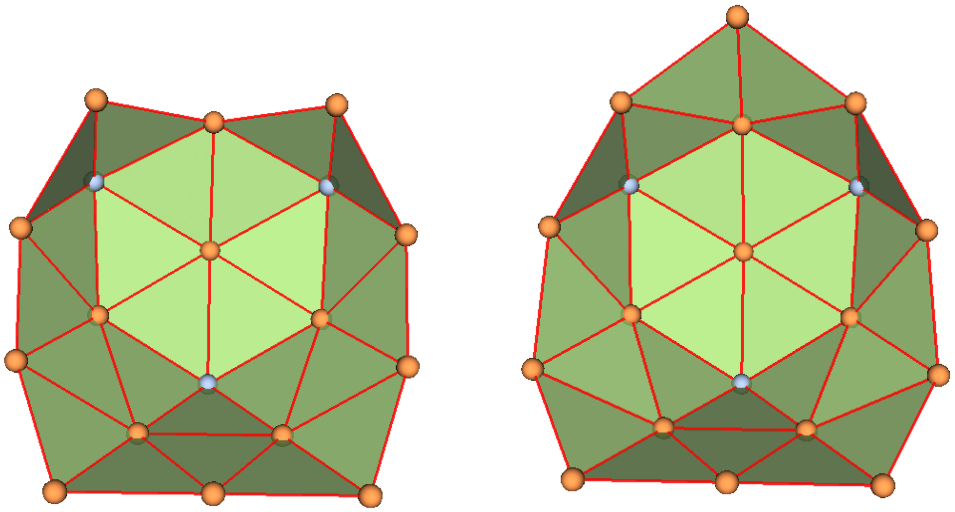
(a) (b)

Figure 11.6.: Two-pentagon IPR pocket region, the $r = 2$ patch in black and the $r = 1$ patch in red. The edges marked by the dotted blue lines were optimized by `edgelength` (a). In (b), the right patch has been positioned the same way, the left patch by a corner merge.

are to short. However this is not a problem, because a fully implemented fill-up algorithm would automatically fix the edge lengths without causing any more geometry problems.

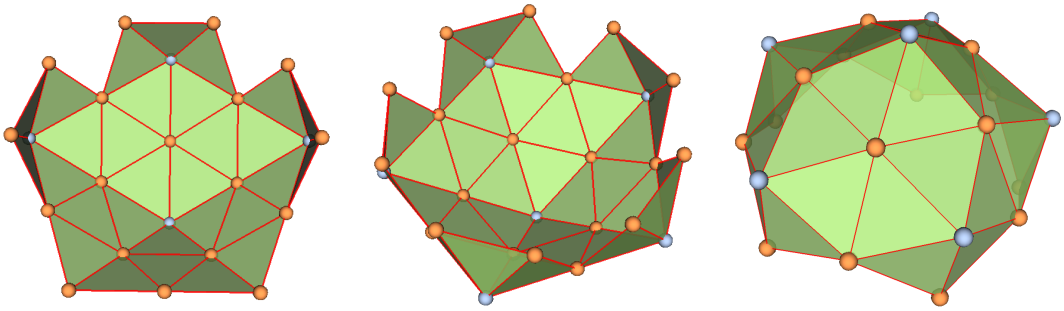
Four to Six Pentagon Nodes

Only a few examples of IPR pocket regions with four to six pentagon nodes were generated. Three interesting examples are shown in Figure 11.8. First, 11.8a is the IPR-equivalent to the rhombus-shape arrangement that we encountered in Figure 11.2a, made fully of corner merges. 11.8b is the IPR-equivalent to the six-pentagon non-IPR pocket region in 11.4, in the sense that both are constructed with corner merges. Meanwhile, 11.8c offers a new possibility of constructing a maximally symmetric six-pentagon pocket region, using only sector merges. All of these pocket regions have flawless geometry.



(a) (b)

Figure 11.7.: Symmetric IPR 3-pentagon pocket region from two sector merges, geometry fixed by `symm_pointdist` (a). Fill-up algorithm can be applied to obtain one additional node position (b).



(a) (b) (c)

Figure 11.8.: Three IPR pocket regions featuring four pentagons arranged in a rhombus with three corner merges (a), a six-pentagon maximally symmetric pocket region from five corner merges (b) and its equivalent from flat merges (c).

12. Conclusion

This thesis project started developing a method for embedding Fullerene surface manifold regions into three-dimensional space. The purpose of such embeddings is to enable a mixed-dimensional treatment of the electronic structure in the molecule surface, combining a two-dimensional, coordinate-free surface DFT with three-dimensional embeddings to improve the accuracy of the model. The achievements of this thesis concerning the construction of embeddings can be summarized by two main points:

1. Developing a method to embed surface manifold regions in three main steps, mainly done in chapters 3 and 5, some directions for future work were explored in 10.
2. Writing an implementation and testing its capabilities for small Fullerenes. The implementations of the three steps were adressed in chapters 8, 9 and 10, chapter 11 presented an overview of the resulting geometries.

It was found that the embeddings are most easily constructed in the *dual representation*, which consists of equilateral triangles, each representing a node of the surface's graph structure or atom in the physical molecule, and given a 3D embedding of the dual representation, the cubic representation's embedding is easily recovered. Compared to the *cubic representation* of the molecule, this is easier to work with, but in turn further away from the physical reality.

The construction of embeddings needs to be started simultaneously from all twelve pentagon nodes, which have been divided in groups of close neighbours that build a high curvature region. The simultaneous start ensures an even shape of the pocket region embeddings later on. By constructing the patches around each pentagon node in the *construction step* with radii such that there are overlaps, the merging was prepared. The *combination step* handles patch merging, where each patch group are merged together into a single pocket region. The *geometry fixing* mechanism makes sure that there are no geometry flaws in the pocket regions after this step. As last one of the three main steps, the *growing step* is responsible for finishing the shape of each pocket region, such that has a layered structure and can be grown to any desired size.

Status of The Implementation and Concept

Neither the method development nor the implementation are finished. The missing piece is the growing step, which was only implemented in parts, such that the single node solver could be applied to a few fitting situations. Also the method development did not

get to the end stages of the growing step. Here, the corner and sector structure of pocket regions, which were discussed in chapter 10 needs to be investigated. This structure can potentially be the key to elegantly solving, or at least reducing the dimension of the systems of equations connected to adding new nodes and new layers to a pocket region. Connected to this is also the center selection for pocket regions, which is needed to define a layer structure on a pocket regions. An initial implementation, selecting the node most centered w.r.t. all pentagon nodes in the pocket region, exists, but it has shown that a single node can not always be the fully symmetric center of a pocket region. Therefore, a more general method that includes centers interior to edges or triangles will be necessary to complete the method. The same holds for the pentagon grouping algorithm and some other parts of the current implementation.

Overall, I would say that the goals that were set in the beginning of this thesis were not completely reached, as a full initial implementation including the growing step would have been the optimal outcome. However, a full implementation of the growing step could in theory be a very powerful tool, for example w.r.t. creating full embeddings of the molecule. It is therefore beneficial to invest more time investigating a well thought-out solution. Creating full molecules could also give an insight on how to construct isometric embeddings of convex polyhedral metrics. Alexandrov's theorem predicts such an embedding to exist for each of these metrics, but finding the embedding for an arbitrary metric is an unsolved problem.

Outlook

Constructing pocket region embeddings is just the first one of a series of problems to solve, if one wants to make a mixed-dimensional treatment of electronic structure on Fullerene surfaces a reality. Assuming that the embeddings can be constructed, the next steps are to compute which embeddings are necessary and which size they should have. Then, it is necessary to connect the three-dimensional embeddings to the two-dimensional surface-DFT, such that the contributions of out-of-surface interactions can be added to the initial calculations to increase the accuracy of the model.

A. Merging Operation

The complete merging process is done by `mergeFaces`, which is shown in listing A.1. The interior of this function will be analyzed now.

```
def mergeFaces(self, newPatchID, overlap):

    newPatch = self.patches[newPatchID]

    #extract the necessary raw data that is to be modified
    newPos = newPatch.positions.copy()
    newTriangles = newPatch.triangles.copy()

    #indexShift: map patch index to pocket index
    indexShift = np.zeros(newPatch.Positions.shape[0]).astype(int)
    nNew = 0

    for i in range(indexShift.shape[0]):
        graphNode = newPatch.global_index[i]

        if graphNode in self.global_index:
            pocketNode = fcs.reverseID(graphNode, self.global_index)
            indexShift[i] = pocketNode
        else:
            #Else, create new index
            fmax = np.max(self.triangles) + 1
            imax = np.max(indexShift) + 1
            indexShift[i] = np.max([fmax, imax])

            nNew += 1

    #Update the newTriangles array with the index mapping
    for i in range(newTriangles.shape[0]):
        for j in range(newTriangles.shape[1]):
            idx = newTriangles [i,j]
            newTriangles[i,j] = indexShift[idx]

    #Extend nodes array and self.fixed array
    positions = np.vstack((self.positions, np.zeros((nNew,3))))
    fixed = np.hstack((self.fixed, np.zeros(nNew, dtype=bool)))
    global_index = np.hstack((self.global_index, np.zeros(nNew, dtype=int
        )))

    #Compute new values
    for i in range(indexShift.shape[0]):
        positions[indexShift[i]] = newPos[i]
```

```
41          global_index[indexShift[i]] = newPatch.global_index[i]
42
43      newRigid = fcs.transferRigidity(indexShift, newPatch.rigid)
44      rigid[newRigid] = True
45
46      #Update original arrays
47      self.positions = positions
48      self.fixed = fixed
49      self.global_index = global_index
50
51      #Update triangles
52      delete = np.zeros(newTriangles.shape[0]).astype(int)
53      for i in range(overlap[:,0].shape[0]):
54          idfier = np.any(newTriangles==overlap[i,0,0], axis=1) * \
55          np.any(newTriangles==overlap[i,0,1], axis=1) * \
56          np.any(newTriangles==overlap[i,0,2], axis=1)
57          delete = delete|idfier
58
59      newTriangles = newTriangles[1-delete == 1]
60
61      self.triangles = np.vstack((self.triangles, newTriangles))
62      self.dual_neighbours = fcs.generateDualNeighbours(self.Triangles)
```

Listing A.1: `pocketRegion.mergeFaces` is the method that merges node positions and triangles of a new patch into the pocket region.

Before starting on the computations, it is necessary to take a look at the inputs. This is a method of the `pocketRegion` object, so the `self` reference in the input creates access to all of the `pocketRegion` object. The actual inputs are `newPatchID` and `overlap`. The latter is known already from the patch positioning chapter and rotation edge selection. The `newPatchID` parameter references, which patch is to be merged. All patches that are grouped into a pocket region, have been added to the `pocketRegion.patches` list, and `newPatchID` references a patch by its index in that list.

The first important step is creating an array that holds the pocket region layer indices for the patch nodes. That happens from line 10 to 22 and the array is called `indexShift`. The index `i` represents the patch index layer, whereas `shiftIndex` is filled with indices from the pocket region index layer. As pointed out in chapter 6.2, to connect these layers, one has to go through the global graph layer, which can be done through the `global_index` attributes of patch and pocket region. Therefore, the global index corresponding to `i` computed in line 14 of the method and saved in `graphNode`.

Now there are two possibilities: If `graphNode` appears in `global_index` of the pocket region, accessed by `self.global_index`, the corresponding pocket region index must be filled into `indexShift[i]`, which is done by the `reverseID` function in line 17. Otherwise, if `graphNode` doesn't appear in `self.indexID`, a new index has to be created for the pocket region. This happens in line 21, where the algorithm decides between `fmax` and `imax`. `fmax` is the maximum value of the `self.triangles` matrix plus one, in other words it is the maximum index of the pocket region index layer plus one. As long as the current new node is the first new node that has been added to the pocket region, this will be selected. However, if the

current node is not the first new node added, then `fmax` is not the correct index, because `self.triangles` is not updated yet. The only array that is updated, is `indexShift`. And because there has been a new node added to this array before, `indexShift` is guaranteed to have the maximum node as value somewhere. That is when `imax` is chosen as new index. The last thing I want to stress is that in line 25, you can see that the parameter `nNew` counts the amount of newly added nodes. This will be needed later.

In the next block of code from line 28 to 31, the `newTriangles` array, which holds the patch's triangles in the patch index layer, has to be transferred to the pocket region index layer. The direct tool for that is the `indexShift` array, and the index changes are done in line 30 and 31. After all arrays that describe nodes have been extended in line 34 to 36, such that they are able to save information about all the new nodes, it is time to fill in that information. `positions`, `fixed` and `rigid` are updated in lines 39 to 49, `triangles` afterwards. For triangles, the overlapping triangles are already known through the `overlap` array and just need to be deleted from `newTriangles`. Afterwards, `newTriangles` can be added to `self.triangles`.

Bibliography

[1] James Emil Avery. "New Computational Methods in the Quantum Theory of Nano-Structures". PhD thesis. 2011.

[2] James Emil Avery. "Wave equations without coordinates I: Fullerenes". In: *Rendiconti Lincei. Scienze Fisiche e Naturali* (June 2018). Accademia Nazionale dei Lincei, Italian Academy of Sciences. DOI: 10.1007/s12210-018-0717-4.

[3] Gunnar Brinkmann et al. "House of Graphs: a database of interesting graphs". In: *Discrete Applied Mathematics* 161.1-2 (2013), pp. 311–314.

[4] NA Cooling et al. "A low-cost mixed fullerene acceptor blend for printed electronics". In: *Journal of Materials Chemistry A* 4.26 (2016), pp. 10274–10281.

[5] Keenan Crane. "Discrete differential geometry: An applied introduction". 2021. URL: https://www.cs.cmu.edu/~kmcrane/Projects/DDG/paper.pdf (visited on 07/23/2021).

[6] Gilles Dennler, Markus C Scharber, and Christoph J Brabec. "Polymer-fullerene bulk-heterojunction solar cells". In: *Advanced materials* 21.13 (2009), pp. 1323–1338.

[7] Dirk M Guldi et al. "Fullerene for organic electronics". In: *Chemical Society Reviews* 38.6 (2009), pp. 1587–1597.

[8] Michael T Heath. *Scientific Computing: An Introductory Survey, Revised Second Edition*. SIAM, 2018.

[9] Pierre Hohenberg and Walter Kohn. "Inhomogeneous electron gas". In: *Physical review* 136.3B (1964), B864.

[10] C Kepley. "Discovery Of A New Mechanism To Control Allergic Asthma Using Fullerene C70 Derivatives". In: *Journal of Allergy and Clinical Immunology* 129.2 (2012), AB365.

[11] Nikolai Plambech Nielsen. "High-resolution Meshes for Two-dimensional Molecular Surfaces". University of Copenhagen, 2021.

[12] Olga V Pupysheva, Amir A Farajian, and Boris I Yakobson. "Fullerene nanocage capacity for hydrogen storage". In: *Nano Letters* 8.3 (2008), pp. 767–774.

[13] Peter Schwerdtfeger, Lukas N Wirz, and James Avery. "The topology of fullerenes". In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 5.1 (2015), pp. 96–145.

[14] Simon Krarup Steensen. “Wave Equations Without Coordinates”. University of Copenhagen, 2020.