Design and modular self–assembly of nanostructures

Joakim Bohlin

Balliol College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2021

This thesis is dedicated to
Ellen ♡ʖ
for joining me on this adventure

Acknowledgements

First and foremost, I would like to thank my supervisors, Professor Andrew Turberfield and Professor Ard Louis, for giving me this opportunity to begin with, and for all their help, knowledge, and support. Andrew has provided sound and rational advice whenever I have needed it, helping me to ensure the scientific relevance of my work and making sure I focus on what is important. Likewise, Ard has provided much appreciated advice, inspiration, and ideas for new research directions and collaborations.

I would also like to thank my fellow members of the Turberfield lab, past and present. For their help, for a friendly working environment and for excellent beta testing of my creations; Dr. Jonathan Bath, Rafael Carrascosa Marzo, Dr. Erik Benson, Dr. Seham Helmi, Dr. Antonio Garcia Guerra, Behnam Najafi, Dr. Emma Silvester, Dr. Robert Oppenheimer, Catherine Fan, Alma Chapet–Batlle, Sing Ming Chan, Qian Zhang, and Dr. Rana Abdul Razzak.

I am also very thankful to Professor Petr Šulc, at Arizona State University, for his advice and guidance concerning oxView development, patchy particle simulation, as well as oxRNA. The secondment I spent in the Šulc group was a very valuable and productive time. My fellow oxView developers Erik Poppleton, Dr. Michael Matthies, Jonah Procyk, and Aatmik Mallya also deserve gratitude for their hard and excellent work.

Similarly, I would also like to thank Professor Ebbe Andersen at Aarhus University for my secondment in his group and for introducing me to RNA origami. Also, thank you Néstor Sampedro for all the help and valuable discussions.

Finally, I also appreciate the help and input from Professor Jonathan Doye and his group, including Hannah Fowler, Dr. Domen Prešern and others.

On a more personal note, for her undying support (despite my endless ramblings about polycubes and simulations), my beloved wife Ellen Bohlin simply cannot be thanked enough. Thank you for joining me on yet another adventure and for enduring the times I still had to leave you behind.

Of course, also want to thank the rest of my family. My father Ulf, who I am certain would also have loved to study here had he been given the opportunity. My mother Okki, for her creativity, her care, and encouragement. My two sisters, Ann-Marie and Ingela, for inspiring me to travel and to study.

I am likewise grateful for the help and encouragement I have received from Ellen's side of the family. My parents-in-law Karl-Johan and Ann-Louise are like an extra set of parents in their support, but I also want to specifically acknowledge Ellen's late grandfather, Anders Bohlin. A botanist in the spirit of Linnaeus, he kept on learning and teaching until the very end.

Our time in Wheatley would not have been the same without such wonderful neighbours; thank you Bruce, Helmer, Harriet and Peter, for making us feel at home and for all the fun we managed to have despite the pandemic.

Finally, I also want to thank our taido and karate friends for keeping us sane and in relative shape during these exciting years: Gunnar and (once again) Rafa for Balliol taido, Emily and Asami for welcoming us to the Exeter club, Jesper and Malin for visiting us in Oxford and through zoom. Also, everyone at the Wheatley Ryobu-kai karate club for many fun sessions.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765703.

Abstract

As nucleic acid nanostructures grow larger and more complex, new tools and methods are needed to facilitate their design. DNA origami structures, for example, are limited by the lengths of their scaffolds but larger assemblies can be bound together by interactions between multiple components. This thesis presents two projects, each approaching the design of such modular structures at a different level of abstraction.

Project 1 presents the *polycube* self–assembly model, where building blocks assemble stochastically using bindings between complementary patches. The assembly of both 2D polyominoes, as well as 3D polycubes, is considered. First, the mapping between input rules (defining the set of available species) and output shapes is investigated, revealing a clear bias toward low–complexity structures. Frequent shapes also tend to be highly modular and symmetric. Secondly, the reversed mapping is explored, presenting a method to find the minimal rule that assembles a specified output shape. Differences in assembly kinetics between possible solutions are investigated using patchy particle simulations, showing that minimal rules can assemble as well as fully addressable rules.

Project 2 presents a more detailed approach to the design of modular structures and individual modules: the *oxView* toolkit for the design, analysis, and visualisation of DNA, RNA and protein nanostructures. While many other design tools exist, oxView makes it easy to import and connect their designs into complete assemblies. Furthermore, oxView allows for free–form editing and rigid–body manipulation. Designs can then be interactively simulated using the oxDNA model, providing a more intuitive understanding of the resulting dynamics.

In conclusion, the design of self–assembling nanostructures has been investigated on both an abstract and a more detailed level. The presented projects have resulted in tools and methods for creating, simulating and analysing modular structures with minimal complexity, potentially containing building blocks created in multiple design software.

Contents

List of Figures

List of Tables

Glossary

2D, 3D . . . . Two- or three–dimensional, referring in this thesis to spatial dimensions of a self–assembled structure.

AIT . . . . . Algorithmic Information Theory.

Colour . . . . . Used in this thesis to denote a type of interaction patch, a "glue type" for binding together polycube components.

DNA . . . . . Deoxyribonucleic acid.

PDB . . . . . The Protein Data Bank (`https://www.wwpdb.org/`). Used in this thesis to refer to the file format used to save atomic structures.

Polycube . . . A 3D shape consisting of multiple cubes connected by their sides.

Polyomino . . A 2D shape consisting of multiple squares connected by their edges.

RNA . . . . . Ribonucleic acid.

SAT . . . . . Boolean satisfiability.

Species . . . . Used in this thesis to denote a type of cube allowed by the polycube rule.

TEM . . . . . Transmission Electron Microscopy

1 Introduction

Contents

How do you design and assemble something on the nanoscale? When working with everyday objects on the macroscale, it is easy to pick the building blocks you want and attach them where you want them to be. However, for nanoscale objects, it is difficult to have that level of top–down control. Instead, more success has been had by imitating nature and letting the building blocks assemble themselves. After millions of years of evolution, nature has the advantage. For artificial nanostructue design, there is much room for progress. This thesis will cover novel methods and tools to design such self–assembling nanostructures.

1.1 Background

This project was funded through the *DNA–Robotics*[1] Marie Skłodowska-Curie Innovative Training Network. The network consists of leading European DNA

[1] `https://dna--robotics.eu/`, grant agreement number 765703

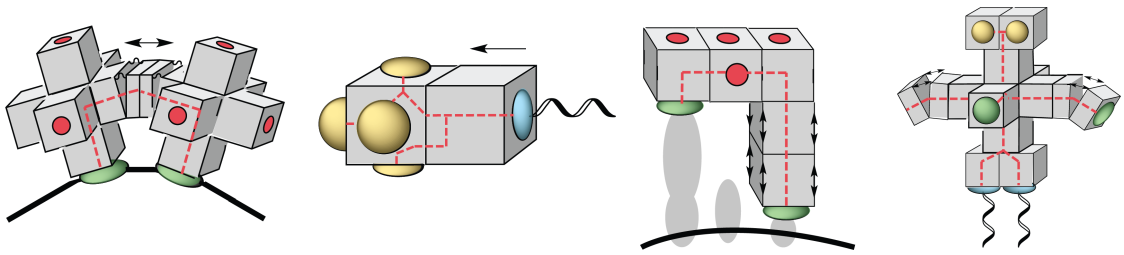


Figure 1.1: The original DNA Robotics design concept. The nanorobots are assembled from standardised cubic modules that functions as sensor, actuators, processors, or structural components. Image adapted from https://dna-robotics.eu/about/

nanotechnology research groups and was formed with the goal of creating a unified framework for integrated biomolecular robotics [1].

This thesis is part of a larger project, with a particular assignment to develop standardised techniques for the design and self–assembly of the nanorobotic modules [2]. The original design proposed for the robotic modules was to use cubic DNA origami modules, as shown in Figure 1.1, but the network also investigated using interacting vesicles as the basic modules. Other researchers within the network are developing such modules with either sensor, actuation or signal processing capabilities. The motivation for developing standardised modules is that, when proven to work, they can be reused in new designs, saving development time and resources.

As such, the contribution of this thesis to the network and the field will be a step further in the ongoing effort to find methods for organising matter on the nanoscale.

1.2 Thesis aim

The aim of this thesis is to present methods that make it easier to design self–assembling nanostructures. This aim partly corresponds to the question of how to find the proper building blocks to use, but also how to simplify the more detailed design of individual components. The next section will cover how the results of these two topics are documented.

1.3 Thesis structure

The thesis covers two related projects, both concerning the design and modular self–assembly of nanostructures. Each project will be introduced by a separate introductory background chapter, with Chapter 2 introducing the first project and Chapter 5 introducing the second.

Modular self–assembly The first project covers results from an abstract self–assembly model called *polycubes*. Chapter 2 provides a background to past experimental and theoretical self–assembly results. Furthermore, it includes an introduction to *DNA origami*, which is also relevant for the second project. Chapter 3 then details the polycube model and the shapes we get when randomly sampling the input space. Chapter 4 presents the results on the inverse problem: given a polycube shape, which input rules will reliably assemble it and what is the least complex input?

Nucleic acid design, simulation, and visualisation The second project takes a more detailed view of self–assembly. Chapter 5 provides background on computer–aided design tools for nucleic acid structures, together with some related simulation models. Chapter 6 presents my contributions to *oxView*, a web–based tool for the visualisation, design, and integration of DNA, RNA and protein structures.

Finally, Chapter 7 discusses the results of both projects and provides some concluding remarks.

2

An introduction to modular assembly

Contents

This chapter will provide a background on modular self–assembly, starting with an explanation of how nucleic acids can be used as a building material and followed by examples of experimentally realised multicomponent structures. The final section will cover relevant self–assembly theory, presenting tile assembly models and their results.

2.1 Nucleic acids as a building material

Professor Ned Seeman was inspired to pioneer the field of *structural DNA nanotechnology* after seeing the woodcut *Depth* by M.C. Escher, where fish are depicted organised into a crystalline structure [3]. As it turned out, the nucleic acid strands of DNA (and later also RNA) can be designed to assemble into similar structures (if the strand sequences were chosen cleverly enough), as this section will explain.

2.1.1 DNA structures

The main building material covered in this thesis is deoxyribonucleic acid (DNA). DNA is a linear polymer best known for encoding the genes of living systems [4]. Strands of DNA are made up of units called *nucleotides*, consisting of a sugar–phosphate backbone unit, as well as one of four possible bases: *adenine* (**A**), *thymine* (**T**) *cytosine* (**C**), and *guanine* (**G**).

Two strands of DNA can bind together to form a helical duplex. Through Watson–Crick base–pairing, the **A** nucleotide form two hydrogen bonds with **T**, while **G** forms three with **C**. Each strand has a directionality, conventionally represented as going from the 3' to the 5' end of the strand (corresponding to the conventional labelling of carbon atom of the backbone sugar), making the duplex anti–parallel.

The base part of the nucleotide is hydrophobic, while the sugar–phosphate backbone is hydrophilic, which means that the bases "hide" on the inside of the duplex to avoid contact with water molecules. However, the length of a backbone unit is about 6 Å (0.6 nm), while the bases would need to be at a distance of about 3.4 Å (the Van der Waals contact separation) to stack stably (without any room for water) [4]. To reconcile this inequality, the DNA duplex forms a double–helical structure with a radius of about 9 Å.

While an unbranched double–helix is the most natural confirmation, it is still possible for duplicates to branch into multiple junctions. For example, the Holliday junction is a junction between four double–helical arms, shown in Figure 2.1 in one of its possible configurations.

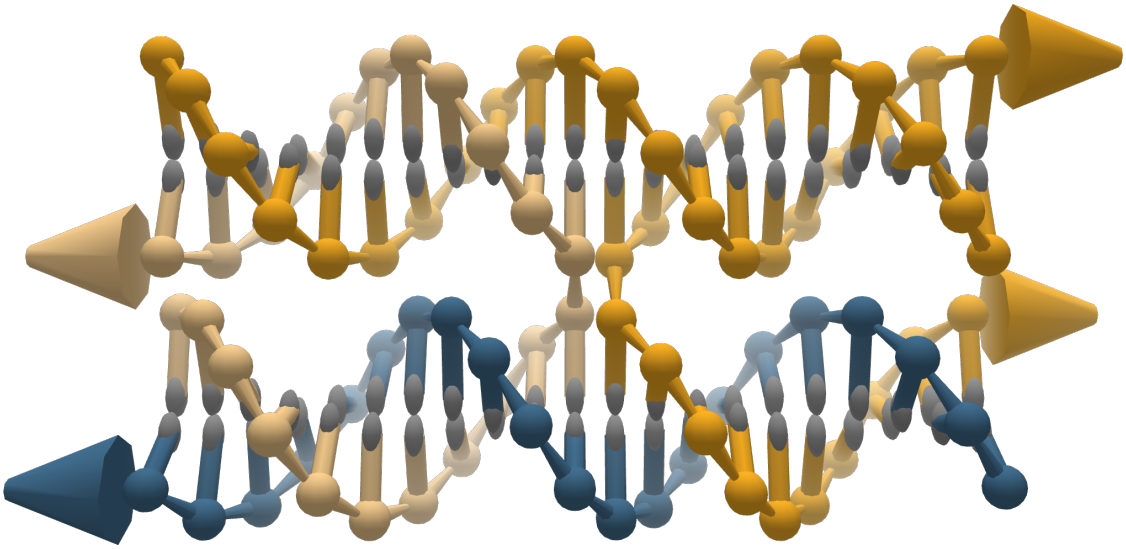


Figure 2.1: Holliday junction, designed in oxView. Sugar–phosphate backbone unit are represented as spheres, while bases are represented as gray spheroids (here without any visual distinction between base types). Arrow heads at the end indicate the 3' end of each of the four strands.

By designing sequences with complementary domains corresponding to intended duplex regions, it is possible to create many different DNA motifs and structures [3, 5], an understanding that pioneered the field of structural DNA nanotechnology and the use of DNA as a building material.

Another breakthrough in the field was the DNA origami technique [6], a now popular and proven method for creating larger irregular structures using DNA. The principle behind it, as illustrated in Figure 2.2, is to use short staple strands, each with one or more domains that are complementary to spatially separated domains on a viral scaffold strand. As the temperature is gradually lowered, the staples bind to the scaffold as seen in Figure 2.2. If done correctly, this "folds" the scaffold into the desired structure, hence the name "origami". However, care must be taken as to how the scaffold should be routed through the design to avoid kinetic traps from where the folding cannot be completed.

With improved design software, it is becoming easier to design DNA origami structures of any given form. See Section 5.1 for an introduction to such tools.

In DNA origami, the size of the structure is limited by the length of the scaffold, which motivates researchers to investigate approaches with multiple origami modules.

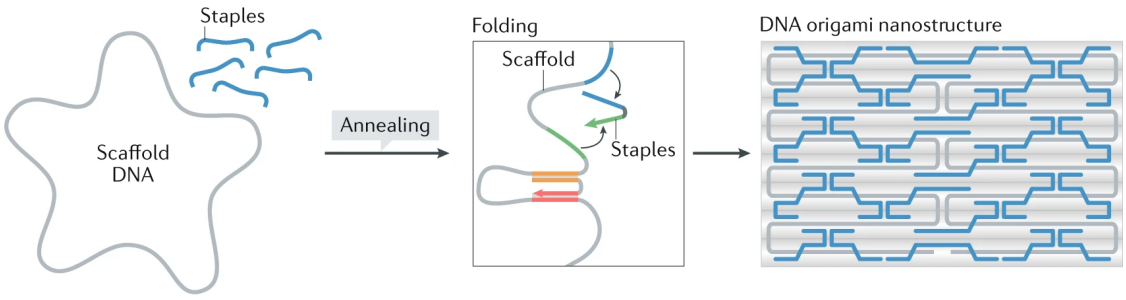


Figure 2.2: Illustration of *DNA origami* [6], adapted from [7]. A long scaffold strand obtained from a virus is folded into the desired nanostructure by multiple short staple strands binding to complementary domains of the scaffold.

Some previous results will be described in Section 2.2. The work described in this thesis aims to simplify the design process significantly.

2.1.2 RNA structures

Another promising building material for self–assembly is ribonucleic acid (RNA). RNA is very similar to DNA, but with a backbone containing the sugar ribose, instead of the deoxyribose sugar found in DNA, and with the *thymine* base (**T**) replaced by *uracil* (**U**).

Biologically, DNA is transcribed into RNA by the RNA polymerase enzyme as part of gene expression [8]. While DNA folding is easier to predict, the more reactive RNA backbone offers more chemical functionality and potential for structural complexity [9].

In 2014, Geary et al. from the Andersen lab in Aarhus, demonstrated a method [10–12] for co–transcriptionally folded RNA origami, which is designed to enable folding *in vivo*. As shown in Figure 2.3, the design used a set of tertiary RNA motifs (such as kissing hairpins and double crossovers) as modules. These modules are combined to generate a blueprint for a single strand, embodying the desired self–interactions. Finally, an iterative algorithm is used to find a sequence that should co–transcriptionally fold into the intended shape.

The Andersen lab is one of the partners of the ITN network I am part of, and I have spent a two–month secondment there working with their RNA origami method. Some of my results on simulating RNA designs are covered in Section 6.7.

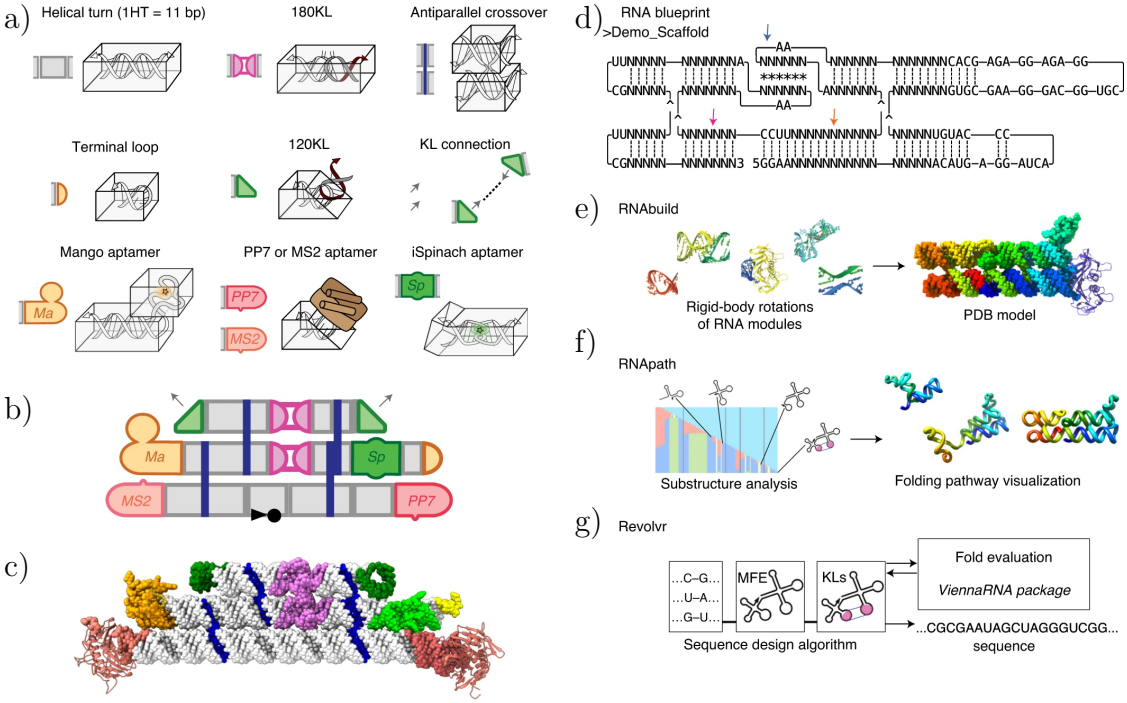


Figure 2.3: Co–transcriptional folding of RNA origami, adapted from [12]. **a)** The set of RNA motifs used as modular building blocks. **b)** Schematic of the modules connected to form a single strand. **c)** Atomistic model of the design in b). **d)** Shows the text–based blueprint used to create designs, while **e)**, **f)**, and **g)** shows scripts developed to aid the visualisation and preform sequence design for the origami.

2.2 Examples of modular nucleic acid structures

From small tiles made from a handful of strands to megadalton–scale structures made from multiple origami designs, modular self–assembly has seen considerable experimental research. This section provides a quick overview of some results of particular interest to the polycube model presented in Chapter 3.

2.2.1 DNA tiles

Following early nucleic acid multi–arm junctions and lattices suggested by Seeman [13], Winfree [14, 15] used double–crossover (DX) motifs, shown in Figure 2.4.a, to self–assemble 2D DNA crystals. The tiles attach using so–called *sticky ends* where, as seen in Figure 2.4.a, one of the strands continues past the end of a duplex region. Each tile has four such sticky ends, so it can connect to four other tiles (as seen in Figure 2.4.b) if their respective sticky end regions have complementary sequences.

Also seen in Figure 2.4.b, the lattices could be made with varying complexity, exemplified using either two or four different species of tiles.

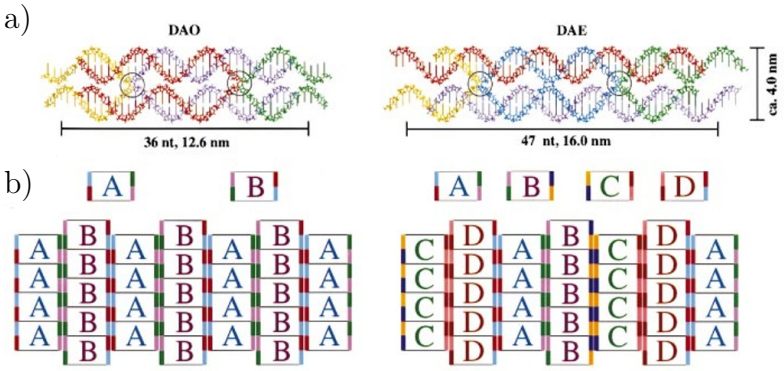


Figure 2.4: DX tiles forming 2D lattices, adapted from [15]. **a)** Examples of tile designs with double–crossover motifs. **b)** Lattices that are made using two and four tile species, respectively.

2.2.2 RNA tiles

As already covered in Section 2.1.2, it is possible to co–transcriptionally fold *RNA origami* [10]. This was first shown in 2014 by Geary et al. who folded RNA tiles that connect through complementary 120-degree kissing loop interactions, as seen in Figure 2.5, forming a hexagonal lattice. A significant promise with co–transcriptionally folded RNA structures is that they can be assembled *in vivo* [9], with a DNA gene being transcribed into RNA, which folds inside a cell.

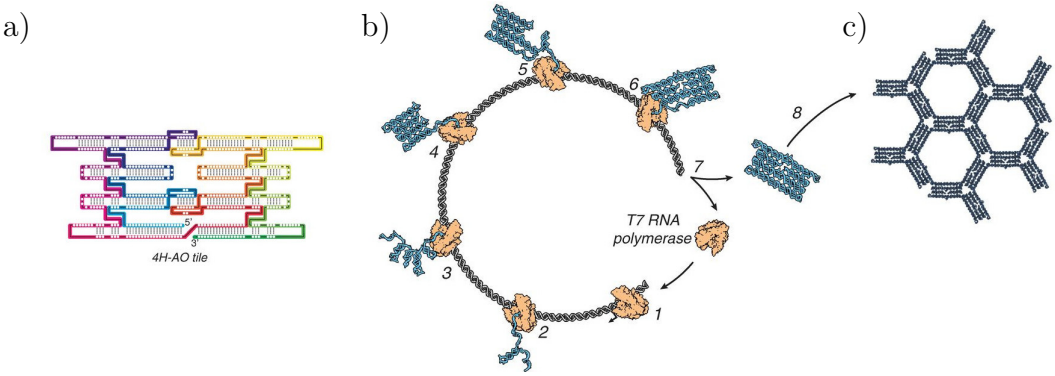


Figure 2.5: Co–transcriptional folding RNA origami tiles, adapted from [10]. The tiles connect through 120-degree kissing loop interactions, forming a hexagonal lattice. **a)** Detailed schematic of the four–helix 4H–AO tile. **b)** Co–transcriptional folding, where the RNA tile folds as it is transcribed from a DNA template. **c)** Hexagonal lattice formed by folded tiles.

2.2.3 DNA bricks

A three–dimensional DNA "canvas" was created in 2017 by Ong et al. [16], using a technique called *DNA bricks*. Structures were assembled from up to about 30,000 unique components, as seen in Figure 2.6. Each "brick" component consists of a single, 52 nucleotides long DNA strand. The strand has four 13-nucleotide domains, each complementary to a domain in a neighbouring brick. A 13-nucleotide helix corresponds to approximately 1.25 turns, creating a 90° dihedral angle between the bricks, as Figure 2.6.a shows.

Since each brick is unique, custom shapes can be "sculpted" by leaving out strands corresponding to voxels (3D pixels) not required.

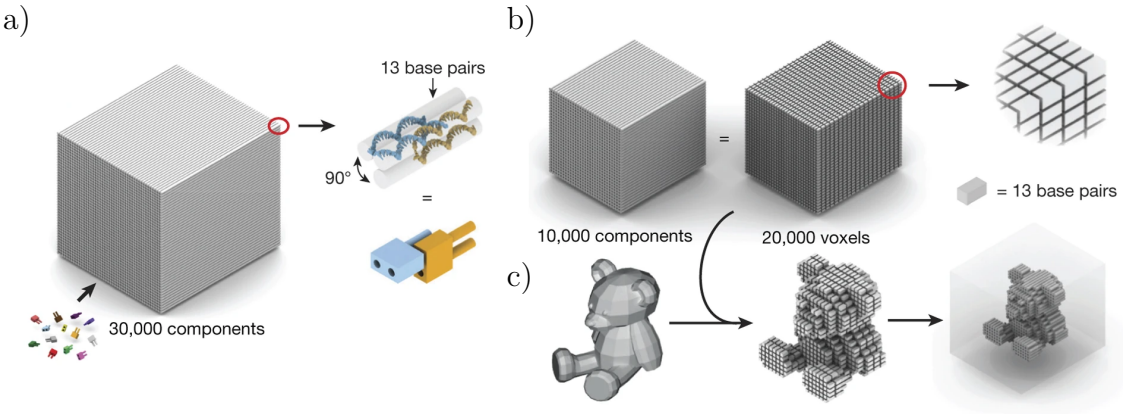


Figure 2.6: DNA bricks, adapted from [16]. **a)** DNA brick structure, where each of the up to 30'000 unique components is a 52 nucleotide DNA strand. The strands connect through a 13 base pair complementary domain at a 90–degree dihedral angle. **b)** A cuboid, here shown with 10,000 components, corresponds to a 20,000 voxel canvas. **c)** Approximating the shape of a teddy bear by removing a subset of the voxels from the canvas.

2.2.4 Finite DNA origami arrays

In 2017, Tikhomirov et al. [17, 18] used the DNA origami technique to demonstrate two–dimensional patterns assembled on the micrometre–scale using square tiles where each tile was a complete origami, as seen in Figure 2.7. The tiles connect through complementary single–stranded overhangs on their edges.

The patterns could either be hierarchically assembled from unique tiles [17] or assembled into random patterns from a small number of tile types [18]. The

binding strength could be adjusted using a variable number of edge overhangs, as seen in Figure 2.7.b). Arrays up to 8×8 tiles were successfully produced, although larger arrays had a much smaller yield (Figure 2.7.c).

While random tilings were generally unbounded, Tikhomirov et al. also showed how to program a finite grid, as seen in Figure 2.7.d). These tiles are very similar to the polyomino model later described in Section 2.3.3.

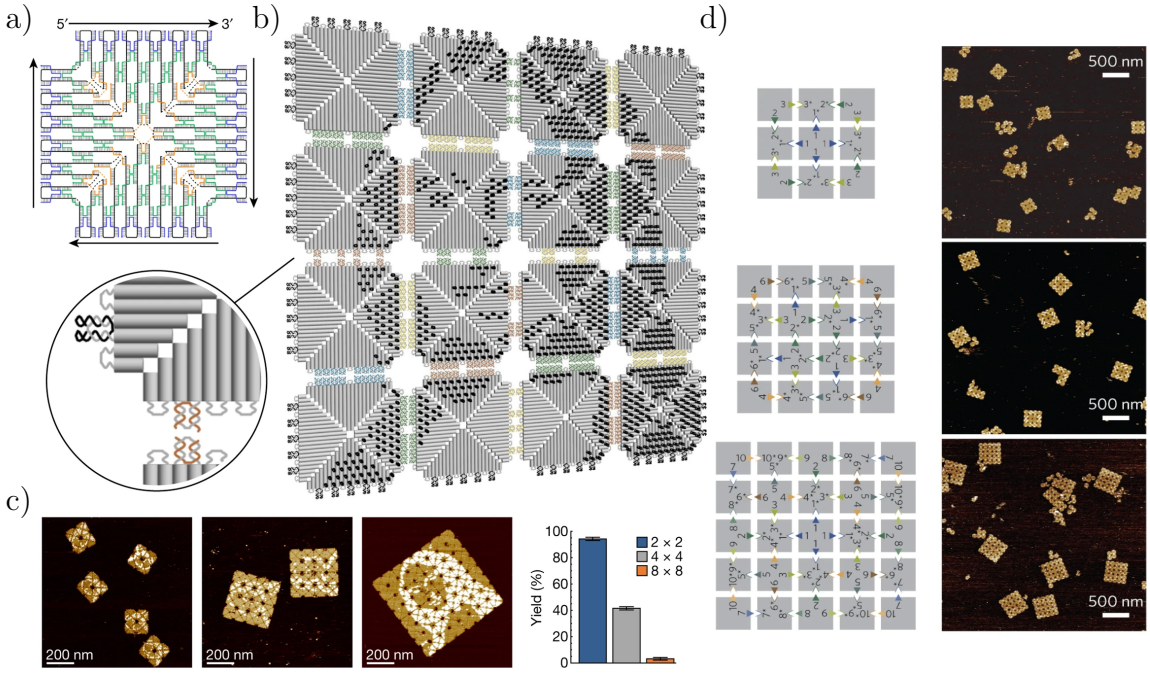


Figure 2.7: DNA origami arrays, adapted from [17, 18]. **a)** Strand–level diagram of a 12×12 version of the the origami tile (actual size is 22×22 helices). **b)** 4×4 tile "Mona Lisa" pattern. The pattern is achieved through double–stranded extensions of selected staple strands inside each origami tile. Staples with the extension correspond to pixels turned on, while those without are turned off. **c)** AFM image of patterned assemblies of different sizes (left) with their respective yields (right). **d)** Abstract design diagrams (left) and AFM images (right) of finite origami arrays, designed to different sizes [18].

2.2.5 Shape–complementary origami

Also in 2017, Wagenbauer et al. [19] used shape–complementarity to assemble DNA origami components into three–dimensional polyhedral shapes up to 450 nanometers in diameter. Later, in 2021, Sigl et al. assembled large shells from shape–complementary origami triangles, as seen in Figure 2.8. The triangular sides attach through protrusions and indentations of complementary shape, as

seen in Figure 2.8.a). By having extra helices protruding from one triangle side, and a hole with a shape they would fit on the side of another triangle, the two components can bind together through stacking interactions at the helix ends, a method first shown by Woo et al. [20].

While the described experiments use triangular components, the general shape–complementary assembly method is a relevant option for a physical realization of the torsionally rigid polycube patches described in Chapter 3.

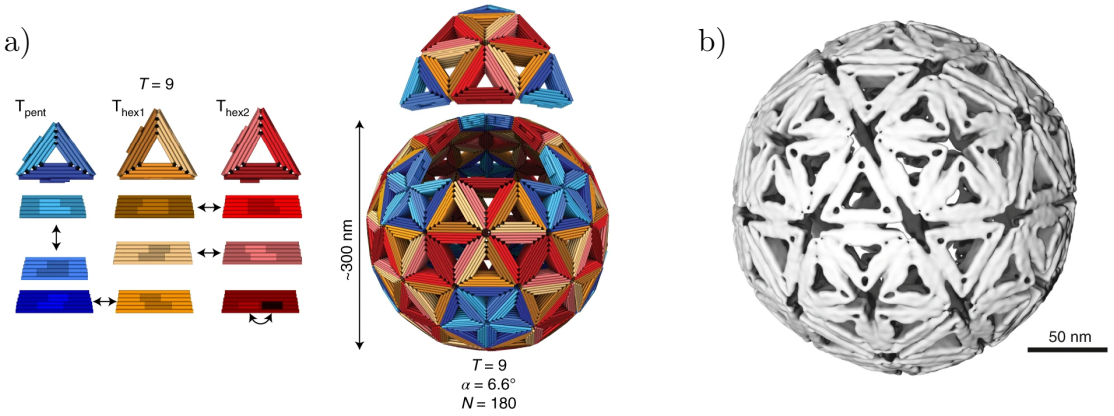


Figure 2.8: Shape–complementary triangles assembling polyhedral shells. Adapted from [21]. **a)** Polyhedral shell design for T=9. N is the triangulation number (the number of unique edges required for assembly), α is the bevel angle of the triangle sides, and N is the number of triangles required for a full shell. **b)** Cryo–EM reconstruction of an assembled T=4 icosahedral shell.

2.2.6 DNA origami nanochambers

In 2020, Lin et al. [22] presented cubic DNA origami "nanochambers". The chambers have sticky–end overhangs on every side of the cube, allowing it to assemble in one, two and three dimensions, as can be seen in Figure 2.9. However, since the shape is only rotationally symmetric around the z–axis, the assembly is still only a limited subset of the polycube model described in Chapter 3.

2.2.7 Octahedral DNA origami frames

In 2020, Wang et al. [23] showed how octahedral DNA origami frames could be used as building blocks in limited and unlimited programmed assemblies, as seen in

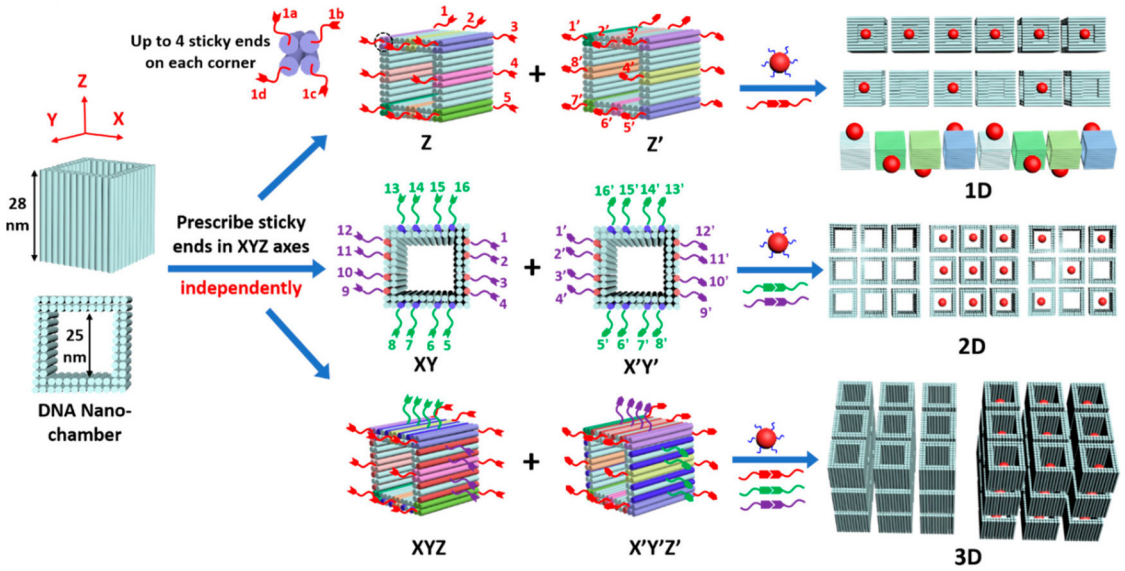


Figure 2.9: DNA origami nanochambers, adapted from [22]. **a)** Concept illustration, where building blocks with *polychromatic* bonds (differentiated though different single–stranded sequences), assemble into 1D, 2D, and 3D structures. **b)** Schematic of DNA nanochamber programmable assembly, showing sticky end overhangs applied in 1D, 2D, and 3D assemblies.

Figure 2.10. Using sticky–end overhangs at the octahedral vertices, the building blocks have the connectivity, as well as the rotational symmetry, of a cube.

Due to the flexibility of the single–stranded connections, the connections are less torsionally rigid than assumed in the polycube model, later described in Chapter 3. However, as can be seen in Figure 2.10, shapes such as the cube can still be assembled with good yield.

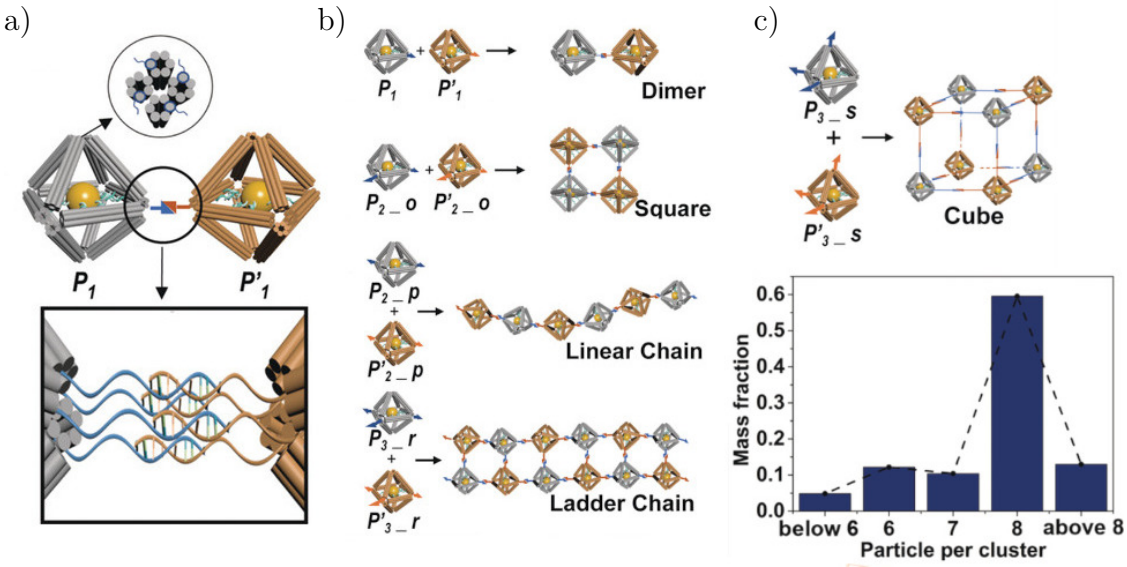


Figure 2.10: Octahedral DNA origami frames, adapted from [23]. **a)** Two octahedra with complementary sticky ends binding together to form a dimer. The edges consist of six–helix bundles. **b)** nanoclusters assembled from different sets of building blocks. **c)** $2 \times 2 \times 2$ cube nano cluster (top) and histogram of the mass fraction, where the intended design of eight components per cluster is the most common.

2.3 Theory of modular self–assembly

With some experimental background covered, let us now look at the progress on the theoretical side of self–assembly design. Although the designs covered are built from nucleic acid strands, it would be computationally impractical to include that level of detail in a self–assembly model. A more straightforward approach is instead to model each component as a discrete tile on a lattice. This section will present earlier such models as a background to my own polycube model, which will be introduced in Chapter 3.

For all modular designs, an important factor is the number of unique components needed. This relates to the concept of complexity covered in Section 2.3.4.

2.3.1 Wang tiles

Introduced by Hao Wang in 1961 [24], *Wang tiles* are square tiles with a colour assigned to each of their four edges. Without rotating or reflecting the tiles, they assemble so that adjacent edges have the same colour.

The DNA tiles by Winfree et al. [15], presented in Section 2.2.1, behave like Wang tiles by design and do not allow rotations or reflections. Winfree investigated the possibility of using such tiles for computation [14], which led to aTAM: the algorithmic Tile Assembly Model.

2.3.2 The algorithmic tile assembly model

The algorithmic Tile Assembly Model (aTAM), shown in Figure 2.11, models the dynamic behaviour of the double crossover DNA tiles introduced by Winfree et al. [15]. Each tile has four patches, one on each edge, corresponding to the four sticky ends of the DNA tile (such as those seen in Figure 2.4). Furthermore, the patches can have different strengths, with a global temperature variable determining the total connection strength required for a tile to attach [25].

In the example seen in Figure 2.11, the pattern grows from the initial bottom–right seed into the blue horizontal bottom row and the rightmost vertical column. This is because these tiles have "strength-2" glues with enough binding strength to attach by themselves [25]. The additional tiles have weaker "strength-1" glues (illustrated as thinner black connectors), so they need at least two complementary patches to achieve the binding strength threshold set by the assembly temperature.

Because of this so–called *co–operative binding*, the tiles can be seen as logic gates performing computation; given the south (bottom) and east (right) patches as input bits, the matching tile attaches and produces two computed output bits as the north (top) and west (left) patches.

2.3.3 The polyomino model

The main inspiration for the *polycube* model (presented in Chapter 3) is the polyomino model [27, 28]. As noted in Section 2.2.4, the 2D poliomino model is similar in assembly to the later experimental micrometer scale tile designs by Tikhomirov [18] shown in Figure 2.7.d). Compared to the aTAM model described in Section 2.3.2, polyomino tiles are allowed to rotate (but not invert),

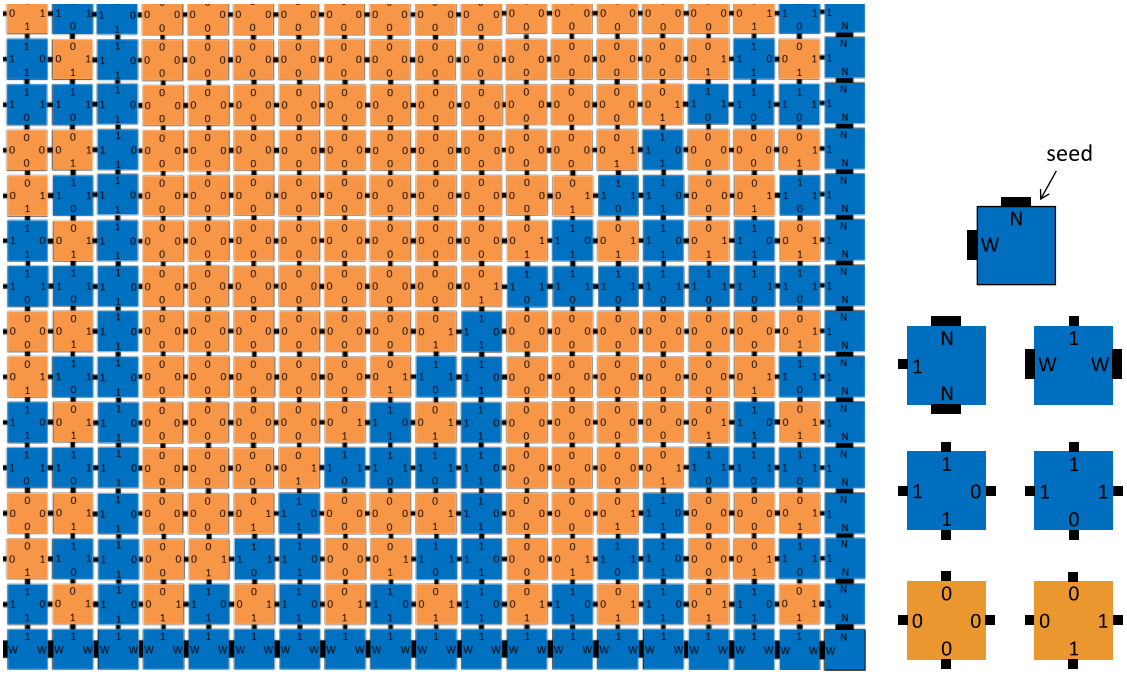


Figure 2.11: Algorithmic self–assembly of a Sierpiński triangle. Adapted from [26]. A tile set (right) grows from an initial seed by co–operatively attaching self–complementary edges (without rotation). The 0 and 1 "glues" are weaker and require two matching bonds to attach (co–operative binding), compared to the W (westward) and N (northward) glues that are strong enough to bind alone.

creating further possibilities for symmetries. Also, the edge binding is not self–complementary, with complementary colour pairs used instead. Finally, polyominoes have a constant binding strength, assembling irreversibly and without co–operative binding (corresponding to "strength-0" glues).

See Figure 2.12 for an illustration of the model, where an input *genotype* (describing the four possible tile types) is mapped into an assembled output polyomino phenotype by stochastically growing the shape from an initial seed. The growth stops if, as in the figure, no more tiles can attach (since the colour 0 does not bind to anything). If the growth is infinite, the genotype is called *unbounded*. A genotype is considered *deterministic* if it assembles the same phenotype polyomino every time. Only output corresponding to bounded and deterministic genotypes are considered *valid*.

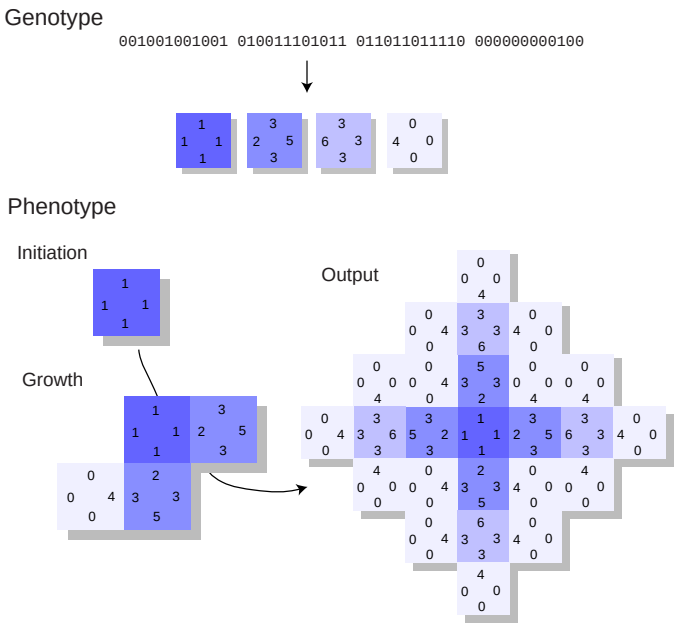


Figure 2.12: Illustration of the polyomino assembly model, adapted from [28]. A *genotype*, in the form of a ruleset of possible tiles, encodes for a polyomino *phenotype*, grown stochastically from an initial seed tile. The integers on the tile edges represent the colour of that edge, where every even integer n binds to the odd $n-1$. The exception is tile edges with the colour 0 that does not bind at all. The output phenotype is grown from an initial seed, a tile of the first species defined in the genotype. Additional tiles, from any species in the genotype, are then added wherever there are compatible edge colours.

2.3.4 Algorithmic Information Theory and input-output maps

If we have a self–assembly model mapping from an input set of building blocks to an output shape, can we find the simplest input for a given output? To answer this, we first need to consider what we mean by "simplest"; what is the *complexity* of a shape?

Complexity has different definitions in different fields, but here we focus on the amount of information needed to describe something, in this case, a shape. Some things, whether in the form of a shape, a song, or a binary string, clearly require less information to describe than others, and we would then call those less complex than their counterparts, but how can we quantify that difference?

Let us first consider the complexity of text strings. If you have a monkey pressing random keys on a typewriter, you would expect it to produce every string of length N with equal probability (assuming the keystrokes were indeed truly random). With k keys on the keyboard, the probability for any string of length

N is then k^{-N}. For example, the title of this thesis, while unlikely to appear randomly, would be equally as probable as any other 50-character string, see the three example strings below, all with probability k^{-50}:

```
DESIGN AND MODULAR SELF--ASSEMBLY OF NANOSTRUCTURES
SHWDRVWKFORWJDOEXOZLSBNREKC Z  VSDJJF  ROKFYRVMIUI
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

But what if we described our strings using algorithms, rather than a full listing of the letters it contains? For example, string number three above could, similarly to the others, then be described using the C programming language as the algorithm:

```
printf("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

But now, a much shorter description also exists:

```
for(int i=50; i--;) printf("A");
```

There are also a number of possible valid variations of the code above, with different variable names and coding conventions, all producing the same output. In other words, not only is the description shorter than writing out the full string but multiple inputs map to the same output, increasing the probability of that particular output further. The arguments above capture the intuition that a string repeating a single letter, or having some other pattern, should be less complex than a random alternative with no shorter description than simply printing the string.

The intuition above can be formalised within the field of Algoritmic Information Theory (AIT), where the central concept is the Kolmogorow complexity (or Solomonoff–Kolmogorov–Chaitin to give full credit), which is defined as the shortest possible computer program to describe a string [29]. More specifically, the Kolmogorov complexity $K(x)$ of an output x is the length of the shortest program that generates x on a Universal Turing Machine (UTM) [29].

AIT leads to many interesting fundamental results. For example, while technically Kolmogorov complexity $K_U(x)$ is always defined for a specific UTM U, because any UTM can simulate any other UTM at constant information cost, asymptotically, the choice of reference machine washes out, and we speak simply of $K(x)$. This

concept is called the invariance theorem. Another important quantity in AIT is the algorithmic probability $P_U(x)$, defined as the probability that, upon randomly selected input programs, a UTM will generate the output x and then halt. There are some technical requirements for such UTMs (for example, they must be prefix machines, meaning that no code contains another code when reading from start to finish). Following a similar invariance theorem, we frequently talk about just $P(x)$, ignoring the choice of UTM. A key property of $P(x)$ is that it can be bounded, as described by the coding theorem, which states that $2^{-K(x)} \leq P(x) \leq 2^{-K(x)+\mathcal{O}(1)}$. In other words, low–complexity outputs are exponentially more likely to be generated by random input compared to high–complexity outputs. This could be compared to how there are many more programs generating the "simple" string number three above compared to the randomly generated string two or the carefully selected string one.

However, a problem with using the coding theorem above is that Komologrov complexity is formally uncomputable due to the halting problem [29]. Fortunately, Dingle et al. [30] were able to derive a computable upper bound to the probability:

$$P(x) \lesssim 2^{-a\widetilde{K}(x)-b} \tag{2.1}$$

where the constants a and b depend on the input–output map used (but are independent of x) and where $\widetilde{K}(x)$ is a computable approximation to the true Kolmogorov complexity. Dingle et al. [30] showed that many different input-output maps follow this upper bound. Moreover, they showed a statistical lower bound, meaning that most randomly sampled programs will generate outputs close to the upper bound [31]. This theorem can be used to connect the probability that a self-assembling shape appears upon a random sampling of rules to the length of the shortest input rule that can make the shape, as will be seen in Section 2.3.5 and Chapter 3.

2.3.5 Evolving polyominoes

In a recent paper [31] the evolution of self-assembling protein complexes was modelled by running a genetic algorithm on the self-assembly of 2D polyominoes.

As illustrated in Figure 2.13, the authors found a strong bias toward structures with low complexity and high symmetry for both systems. In addition, they showed that randomly sampling assembly programmes gave a very similar probability-complexity relationship for the polyominoes as was found for the evolutionary runs. Therefore the simplicity does not arise from natural selection but is instead a property of the mapping, as predicted by the alternate coding theorem (Equation 2.1) of Dingle et al. [30].

The evolutionary fitness of the polyominoes only depended on their size (16–mers had the highest fitness).

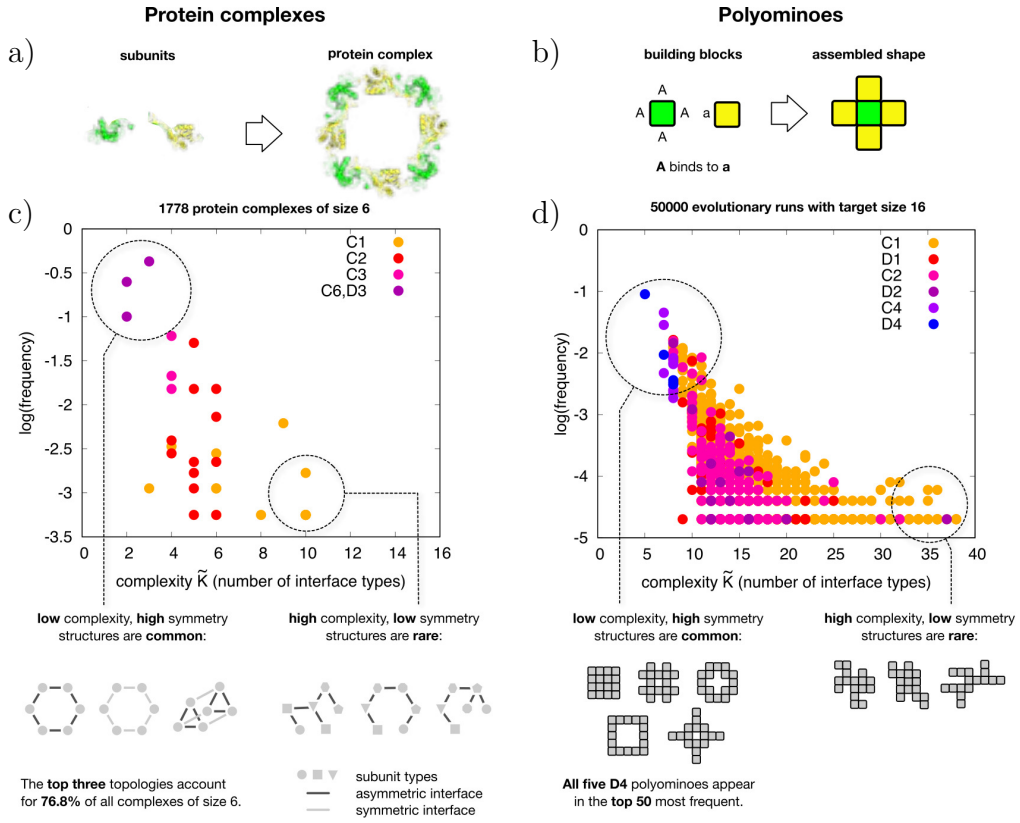


Figure 2.13: Frequent symmetry and simplicity through evolution, adapted from [32]. Both protein complexes (a) and polyominoes (b) self–assemble from individual units. **c)** Frequency of 6–mer protein complex topologies in the protein data bank, versus their complexity (measured as the number of interface types) **d)** Frequency versus complexity of polyominoes found in evolutionary runs with a fitness function seeking 16–mers.

3

Modular self-assembly of polycubes

Contents

The previous chapter introduced several experimental projects and theoretical models concerning the design of multi-component objects. However, while some coarse-grained tile models exist, there remains a need for fast methods to explore the assembly of multi-component 3D structures. This chapter extends the 2D polyomino

model [27, 28] to 3D, which we will call the *polycube*[1] model. The program simulating the 3D polycubes can also easily simulate 2D polyomino assembly. We will start by performing a reference sampling (Section 3.3) to verify that the 2D polyomino model agrees with earlier results. We will then compare 2D and 3D results in Section 3.4 and, in particular, see how the probability-complexity relationships compare to the predictions of the AIT coding theorem.

The following chapter (Chapter 4) will then show how to explore the solution landscape of, and obtain the lowest complexity possible assembly rule for, a given shape.

3.1 The polycube model

A *polycube* consists of multiple equally-sized cubes connected by their neighbouring faces (a three-dimensional analogue to how polyominoes are squares connected by their neighbouring edges). In the model presented here, a polycube is stochastically self-assembled according to a specified *rule*, defining a set of available cube species. Each species describes a type of cube that can be present in the polycube, so cubes belonging to the same species are always identical. See, for example, Figure 3.1, where an input rule with two species assembles into a double-cross output polycube with seven cubes.

Each species has six patches, one on each face of the cube, and each patch has a "colour" and an orientation. The colour is indicated by a signed integer and the orientation is one of four possible rotations:  (0),  ($\frac{\pi}{2}$),  (π) or  ($\frac{3\pi}{2}$), saved as an integer 0, 1, 2, or 3. For each of the patches, facing left $[-1, 0, 0]$, right $[1, 0, 0]$, bottom $[0, -1, 0]$, top $[0, 1, 0]$, back $[0, 0, -1]$ and front $[0, 0, 1]$ respectively, the default (0) orientation () corresponds to the vectors $[0, -1, 0]$, $[0, 1, 0]$, $[0, 0, -1]$, $[0, 0, 1]$, $[-1, 0, 0]$, and $[1, 0, 0]$.

A patch can bind to another patch if and only if they have complementary colours and the same (global) orientation. In Figure 3.1, the patch color 1 is shown as bright red while the complementary -1 is a darker red colour. If the patch

[1] A polycube is the 3D equivalent of a polyomino, consisting of one or more equal-size cubes joined face to face.

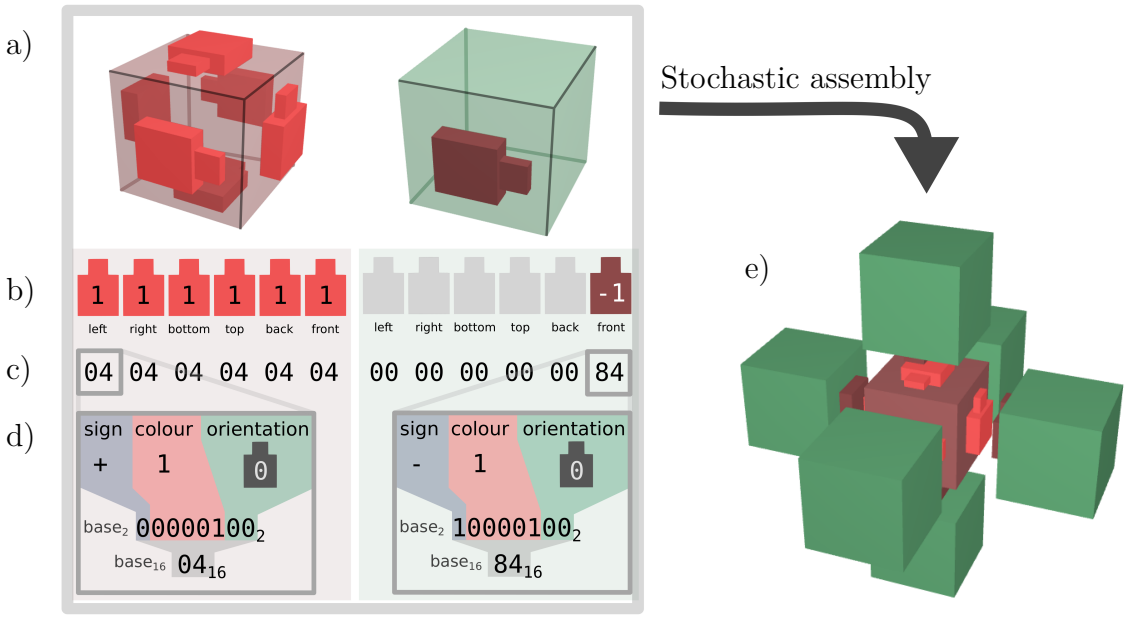


Figure 3.1: Illustration of the polycube model and notation, exemplified with the rule 040404040404000000000084. Compare this to the polyomino model in Figure 2.12. **a)** 3D representation of the species in the rule. **b)** Rule depicted as a list of the patches in each species. The empty patches (colour 0) in the green species are just shown with their orientations. All orientations are 0 in this rule, since changing them would not change the output. **c)** Hexadecimal representation of the rule, shown decoded in **d)**, where every 2-digit hexadecimal number represents a patch. Converted to a 8-bit binary number, first bit encodes the sign, the next five bits the colour ($[0, 31]$), and the final two bits encode the orientation ($[0, 3]$). **e)** Fully assembled polycube output. The assembly used one copy of the first species (red) and six copies of the second (green). The assembly finished since no further cubes could be added.

colour is zero, the patch is shown as empty and will not bind to anything. All opposite colours are defined as complementary, such that n pairs with $-n$. The model could, however, be expanded to more complicated colour interaction matrices or changed to pair odd integers with each subsequent integer as in the polyomino model [27, 28] described in Section 2.3.3.

By constraining the input space not to use the back and front patches, while orienting the remaining patches to point towards the front direction, the output space will correspond to 2D polyominoes.

3.1.1 String representation of the input rules

Polycube rules can be described in a hexadecimal representation, as seen in Figure 3.1.c). Each species is described by 12 hexadecimal digits, two digits per patch. The two hexadecimal digits are then converted into eight binary digits (bits). The first six bits represent the patch colour as a signed integer (allowing for decimal values 0-31); the remaining two encodes one of the four possible patch orientations.

Alternatively, a less compact but more human-readable decimal notation can be used, with each patch written as `c:o` (where `c` is the integer colour and `o` is the integer orientation) and delimited by vertical bars (`|`). Finally, each species is delimited by underscores (`_`). For example, the rule used in Figure 3.1, "040404040404 000000000084", would be written as "1:0|1:0|1:0|1:0|1:0|1:0_|||||-1:0"

3.1.2 Stochastic self-assembly

The stochastic self-assembly of a polycube starts by placing a cube from one of the available species as a seed at the origin. If the assembly mode is *seeded*, it will always use the first species of the rule. If the assembly mode is *unseeded*, the seed species is instead chosen at random. Once a cube has been added to the assembly, each of its yet unbound patches will be added to a list of *moves*, where a move represents a position where another cube can be added.

Moves are then processed until the list of moves is empty, or the polycube grows beyond a specified size, at which point it is considered *unbounded*. Figure 3.2.a) shows an example of an unbounded structure tiling the plane using two species.

While the list of moves is not empty, a random move is chosen at each step. The input rule is then randomly searched for a species fitting the move. Cubes can be rotated to fit, and if a fitting species is found, the corresponding cube is added to the assembly. If there is no fit to be found, the move is discarded.

The assembly is repeated n_{times} times (default 100), and the outputs are compared for equality (allowing rotation) in order to determine if the rule is *deterministic*. Note that this method is not rigorous, but it is enough to quickly

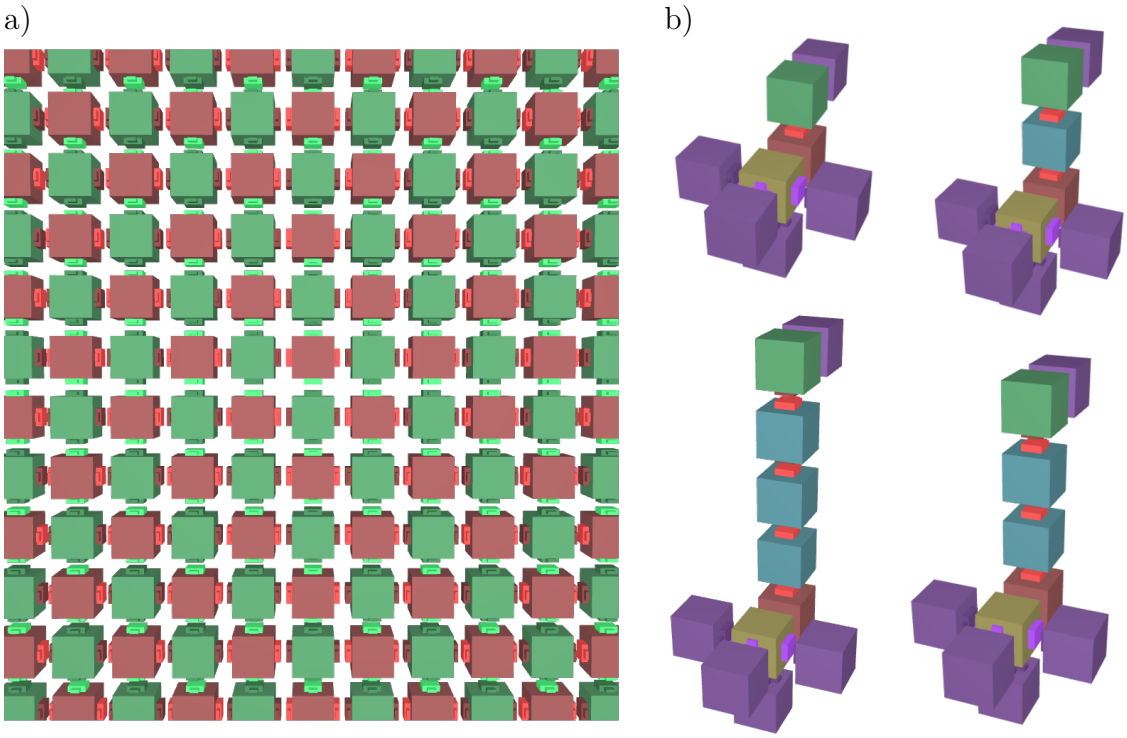


Figure 3.2: Examples of undefined assemblies. **a)** Unbounded assembly that tiles the plane using two species (05050a08000085858a880000), **b)** An undeterminisic assembly of a "giraffe duck" with a neck that can have a different length each time it is assembled (00000006008b00008600000c000000028c00080c0c000c0c048600000000).

detect a large fraction of the nondeterministic assemblies. Figure 3.2.b) shows a non-deterministic structure where the blue "neck" species can bind to itself. Thus, the output depends on how many cubes from the blue species bind before a green species cube stops the growth. Both species are equally likely to bind, so the probability of assembling a neck with l blue cubes is 2^{-l}. Unbounded or non-deterministic structures are considered *undefined* and are not included in the sampling result. This is inspired by the idea that unbounded protein complexes are highly deleterious [32].

It could be argued that instead of first picking a random move and then randomly trying all available species to find a fit, one should pick both a move and a species at random until a fit is found. While this would take a longer time, it would avoid biasing the assembly toward unlikely assembly results, where a move is picked that would otherwise usually be blocked by more likely surrounding cubes. However, since only deterministic and bounded rules are of interest, this would only affect the end result in the cases where the bias is strong enough and n_{times} is low enough

to incorrectly make the rule seem deterministic.

For an illustration of the model, let us return to Figure 3.1. The example in the figure is a three-dimensional "double-cross" structure created from a rule of size 2. The initial seeding cube belongs to the first species, enabling six additional cubes, all belonging to the second species, to bind at each patch. The patches bind since their colours, 1 and −1, are opposites. After all six outer cubes have bound, there are no remaining possible moves, and thus the polycube stops growing. Since the growth stops, this particular polycube is bounded at a size of seven cubes. Furthermore, since the rule gives the same polycube every time it is evaluated, the polycube is deterministic.

3.1.3 Implementation

The polycube assembly model is implemented in two versions: one browser-based implementation in JavaScript (`https://akodiat.github.io/polycubes`) for outreach activities and accessible visualisation, and one C++ implementation for fast rule evaluation. An example screenshot from the JavaScript implementation is seen in Figure 3.3. The C++ code also includes a Python binding for simplified analysis. More details on the code can be found in Appendix A.

3.2 Sampling the space of assembly rules

Having now understood the assembly model, let us use it to explore how the space of input rules maps onto the output shapes.

3.2.1 Sampling vs exhaustive search

First of all, trying every input in a brute-force approach would be unfeasible for most input spaces. For a 3D polycube space with $\widetilde{K}_s$ species and $\widetilde{K}_c$ colours, four possible patch orientations, and six patches per species, we get the following input space size, where $I^{3D}_{\widetilde{K}_s,\widetilde{K}_c}$ is the set of all inputs:

$$\left|I^{3D}_{\widetilde{K}_s,\widetilde{K}_c}\right| = (4 \times (1 + 2\widetilde{K}_c))^{6\widetilde{K}_s} \tag{3.1}$$

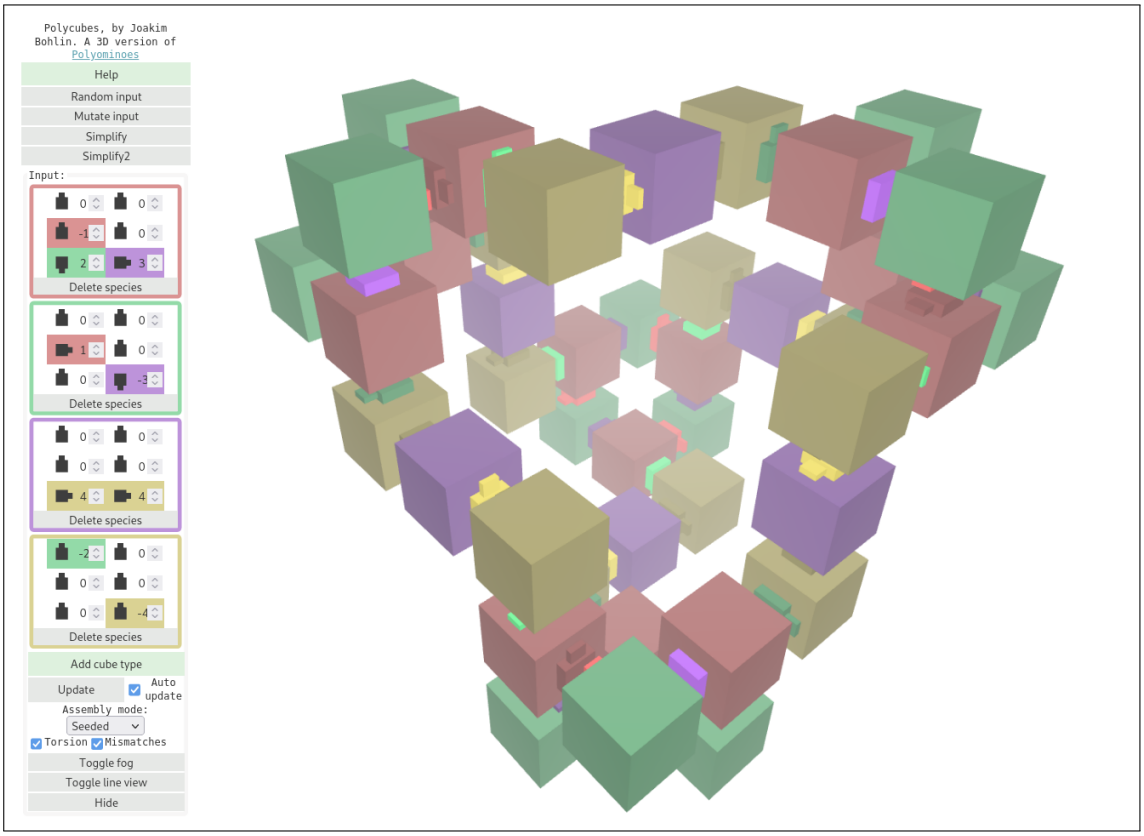


Figure 3.3: Browser-based implementation of the polycube model and stochastic assembler. The species are represented as boxes in the menu to the left, each patch having an input for colour and orientation. The resulting cube is automatically assembled on the background canvas (right). The tool can be accessed at `https://akodiat.github.io/polycubes`

Even for relatively small values of $\widetilde{K}_s = 3$ species and $\widetilde{K}_c = 2$ colours, we get $I^3_{2,3} \approx 2.62 \times 10^{23}$. Both $\widetilde{K}$ measures here can be seen as proxies for Kolmogorov complexity, as discussed in Section 2.3.4.

For 2D, it is a bit easier since there are only four patches per species and a fixed patch orientation:

$$\left|I^{2D}_{\widetilde{K}_s,\widetilde{K}_c}\right| = (1 + 2\widetilde{K}_c)^{4\widetilde{K}_s} \tag{3.2}$$

, but the space still grows exponentially.

However, even if the space of possible input rules is too large to explore fully, we can still get an idea of how likely it is for an input rule to map to a particular output shape through uniform sampling and assembly of a large number of random

rules. The results of these samplings will be covered in the following sub-sections.

3.2.2 Classifying the sampling output

As described in Section 3.1.2, rules growing larger than 100 cubes were discarded as unbounded, while those remaining bounded were re-assembled 100 times in an effort to remove nondeterministic rules. The deterministic and bounded output were then grouped by their shapes, counting the number of times each given shape occurs. For each rule found to produce a given shape, three different complexity measures were recorded:

$\widetilde{K_s}$ - The number of species used.

$\widetilde{K_c}$ - The number of colours used.

$\widetilde{K_{lz}}$ - The length of the binary rule after Lempel-Ziv compression [33].

The Lempel-Ziv compression measure was chosen for its use in previous literature [30, 32] and because of the connection between Kolmogorov complexity and compression [32]. It is defined as in [30], where $N_w(x)$ is the number of patterns (words) in a binary string x (converted from the hexadecimal string representation described in Section 3.1.1):

$$\widetilde{K_{lz}} = \begin{cases} \log_2(n) & : \ x = 0^n \text{ or } x = 1^n \\ \log_2(n)\left[N_w(x_1 \cdots x_n) + N_w(x_n \cdots x_1)\right]/2 & : \ \text{otherwise} \end{cases}$$

Each rule was simplified before the complexity measures were calculated in order to compare them more fairly. This simplification was done by removing all patches without a complementary colour in the same rule, as well as setting the orientation of all empty (zero-coloured) patches to zero. Finally, the colour indices were updated to avoid any gaps in the numbering.

3.2.3 Input spaces sampled

Multiple samplings were done to explore how the input space parameters affect the output shape space. Table 3.1 lists the samplings and their properties, with the following sections discussing them in more detail.

	Polyomino reference sampling	Polycube samplings			
Input space	$I^{2D}_{16_s,31_c}$	$I^{2D}_{5_s,31_c}$		$I^{3D}_{5_s,31_c}$	
Assembly type	Seeded	Seeded	Unseeded	Seeded	Unseeded
Dimensionality	2D	2D		3D	
Number of samples	10^9	10^8			
Max number of species	16	5			
Max number of colours	31	31			

Table 3.1: Samplings of the space of polycube and polyomino input rules.

3.3 Polyomino reference sampling

In order to verify the model against earlier polyomino results [32], a large reference sampling of the 2D space was performed. Limiting the input space to 16 species and 31 colours ($I^{2D}_{16,31}$), one billion (10^9) random rules were assembled using the seeded assembly mode (Table 3.1). The total number of possible rules for the space is $\left|I^{2D}_{16,31}\right| = 63^{64} \approx 1.4 \times 10^{115}$ (see Equation 3.2).

3.3.1 Input space analysis

Let us start by looking at the sampled input space. How much of the input maps to valid output? As can be seen in Figure 3.4, a large number of the sampled rules were either unbounded or not assembling deterministically. The boundary between unbounded and non-deterministic assemblies is not clear, since some input also can be both (but is classified as either one or the other depending on what it assembled as first).

Figure 3.4: Proportion of valid rules when sampling $I^{2D}_{16_s,31_c}$ using seeded assembly. From a total of 1,000,000,000 sampled rules, 44,545,570 were found to be valid, while 871,155,425 were unbounded and 84,299,005 were non-deterministic

3.3.2 Output space analysis

With the input space investigated, we can also check how much of the output space we managed to sample. Figure 3.5 shows that most of the valid rules are assembled into smaller shapes, with over 60 per cent (27 million) being 1–mers (note that the y-axis is logarithmic).

Note how some polyomino sizes that are multiples of four have a much higher count than their neighbours, indicating a bias for shapes of those sizes. Given the polycube patch interaction rules, some structures are much easier make. A single species rule cannot make a 2–mer or a 3–mer, but can easily form a valid 4–mer by making a square. The square can then be made into an 8–mer "Catherine wheel" by introducing a second species attachable to the first (and a 12–mer with a third species, et cetera). Since these shape sizes can be made using fewer species than others, the probability for a random rule to contain such species is higher. This will be later seen in Figure 3.7, where the most common shapes are seen composed of blocks of four identical sub–shapes. This mechanism is similar to what has been seen for evolutionary runs of polyomino shapes [28], where 16–mers evolved through intermediate shapes increasing with four cubes at a time.

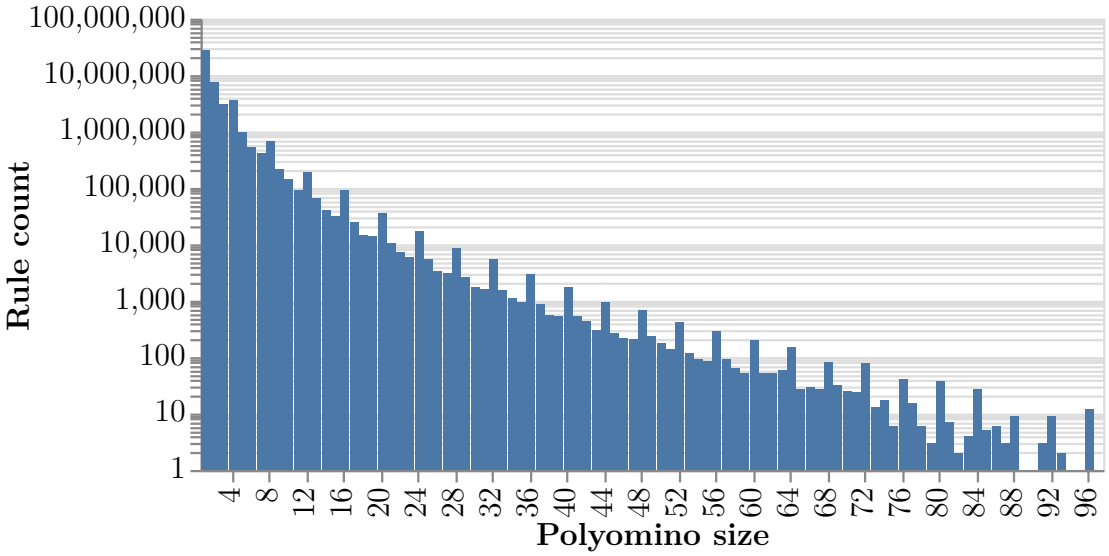


Figure 3.5: Number of input rules per output size. Distribution of polyomino sizes of 10^9 sampled rules in $I^{2D}_{16_s,31_c}$.

In Figure 3.6, the blue bars show how many polyominoes were found of each size, while the red line shows the total number of polyominoes that exist of that size (obtained from the On-line Encyclopedia of Integer Sequences [34, 35]). The sampled input space $I^{2D}_{16_s,31_c}$ should allow for all 16–mers to be assembled (provided enough samples) since there are enough species and colours for fully addressable assemblies (assigning each cube its own species and each connection its own colour). Thus, the blue bars would follow the red line until size 16 if the complete input space was enumerated. Figure 3.6 shows that we find at least the correct order of magnitude up until about 10–mers, after which polyominoes of larger sizes are found in decreasing numbers.

For the larger sizes, we again see the bias toward shapes with a multiple of four tiles. This shows that not only do these polyomino sizes have more input rules assembling them, but that a larger number of unique polyomino shapes are also found for them. The OEIS (On-line Encyclopedia of Integer Sequences) data shows no such bias in the actual size counts. Following the earlier argument that few additional species are required to extend a shape with a multiple of four tiles, those additional tiles can be placed in various (still symmetric) configurations to create different shapes. Larger shapes leave room for more such permutations, leading to the increasing bias seen in the figure. Thus, this is not a sampling artefact but rather an intrinsic bias in the polyomino input-output map.

With the sampling details explained, we move on to the main polyomino results. Let us start by looking at the different shapes found. Figure 3.7 shows a gallery of the 16–mer polyominoes most commonly found when sampling $I^{2D}_{16,31}$. Each shape is scaled in proportion to the frequency at which it was found, showing that, even within a given shape size, some shapes are much more common than others. Note also how the most common 16–mer shapes can be assembled using a small number of species and colours.

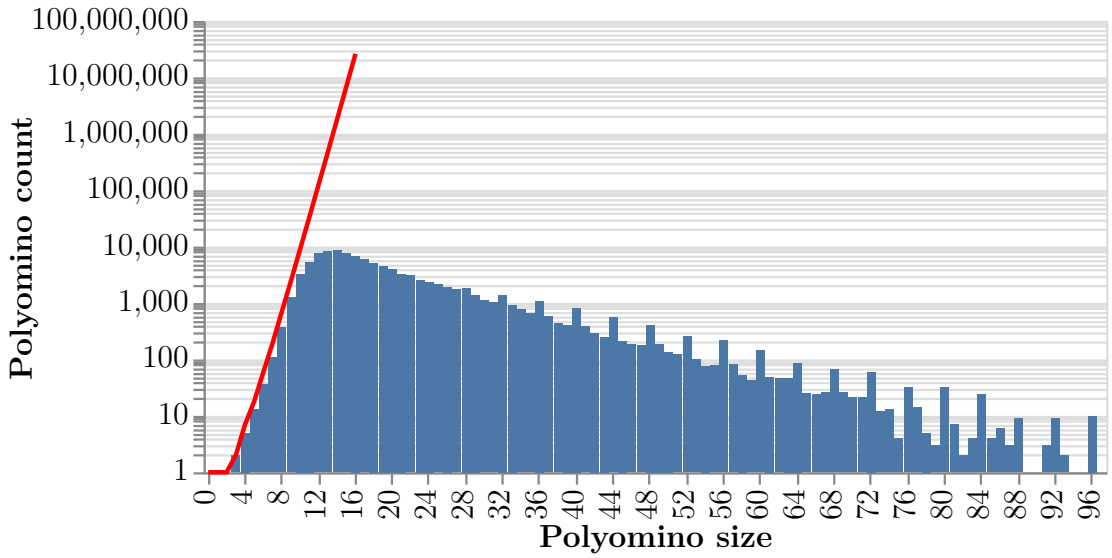


Figure 3.6: Number of output polyominoes per output size. Count of unique output polyominoes found sampling 10^9 input rules in $I^{2D}_{16_s,31_c}$. The red line shows the total number of polyominoes of each size, obtained from OEIS A000988 [34, 35]. For larger sizes, only the most likely structures appear due to sampling constraints.

3.3.3 Symmetry

Noting that frequent 16-mers look more symmetrical, we quantify symmetry by calculating the rotation and reflection symmetries for 2D polyominoes on the lattice. Corresponding symmetries could also be calculated for 3D polycubes, but we focus on polyominoes as their symmetry groups are fewer and since this will enable us to compare the results to those of Johnston et al.

citejohnston2021 seen in Figure 2.13.

A 2D polyomino has four possible rotations, corresponding to the matrices $\left(\begin{smallmatrix}1&0&0\\0&1&0\\0&0&1\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}0&-1&0\\1&0&0\\0&0&1\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}-1&0&0\\0&-1&0\\0&0&1\end{smallmatrix}\right)$, and $\left(\begin{smallmatrix}0&1&0\\-1&0&0\\0&0&1\end{smallmatrix}\right)$. The first one is simply the identity matrix and can be ignored. However, if the lattice coordinates remain unchanged after multiplication with any of the others, it means that the polyomino has rotational symmetry. A polyomino also has four possible reflections: one across the x-axis, one across the y-axis, and two along the diagonals. Similarly, a 3D polycube has 24 possible orientations (including identity), as well as reflections around the x-y, x-z, y-z and diagonal planes.

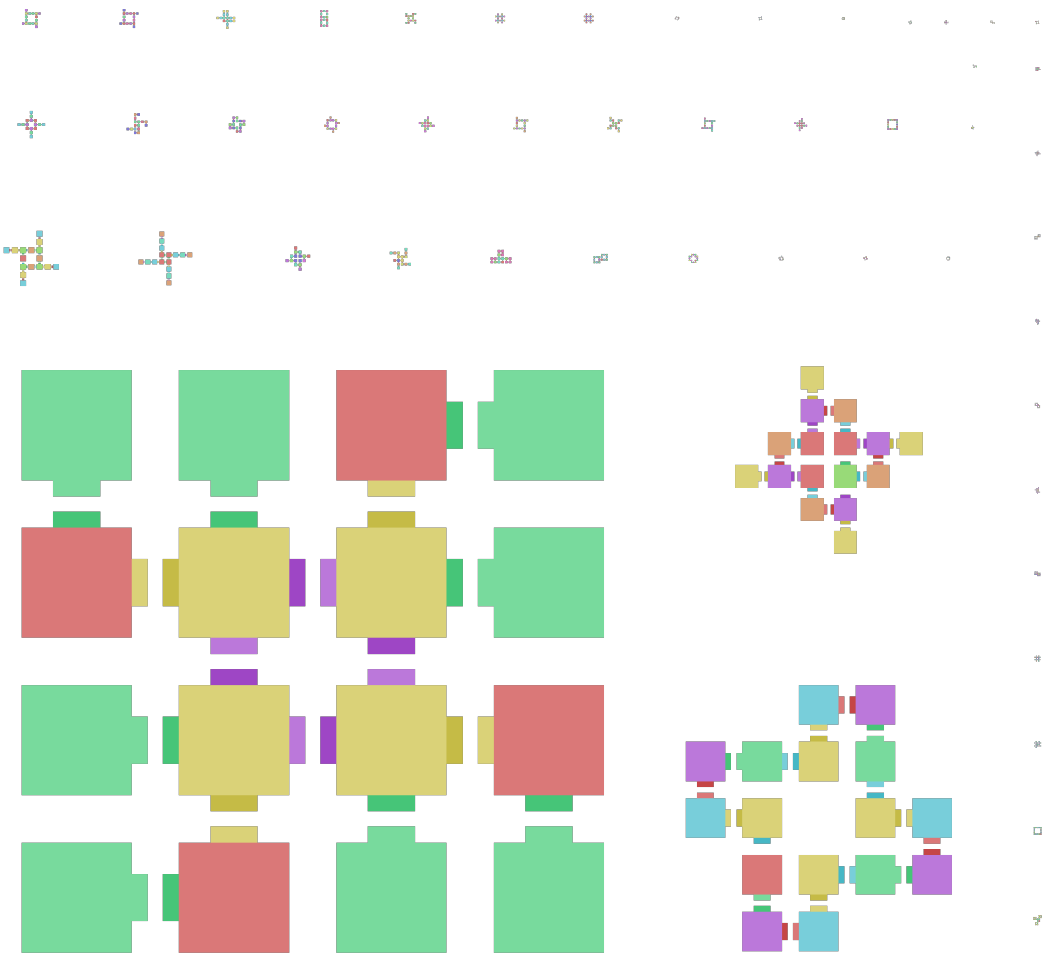


Figure 3.7: The 50 most common 2-dimensional 16–mers, scaled in proportion to their frequency. Found while sampling the $I^{2D}_{16,31}$ input space.

In the simplified context of these lattice shapes, we assign 2D symmetry groups as follows:

```
if reflsymms == 2:
  group = 'D4'
elif rotsymms == 3:
  group = 'C4'
elif reflsymms == 1:
  group = 'D2'
elif rotsymms == 1:
  group = 'C2'
elif rotsymms == 0:
  group = 'C1'
elif reflsymms == 0:
  group = 'D1'
```

3.3.4 Simplicity bias

Consider again the simplicity bias presented in Section 2.3.5. Can we show such a trend for the current model as well? As can be seen in Figure 3.8, there is a log-linear relationship between the probability of finding a rule that assembles into a particular structure and the information needed to specify the structure, as predicted in [30, 31].

Comparing this with Figure 2.13, it is clear that the same simplicity bias is present here. Note that Figure 2.13 shows results from an evolutionary run, but that Johnston et al. also show simplicity bias for sampled data in the supporting information of the article [32].

The unusually low frequency of the two C2 polyominoes with $\widetilde{K}_c = 4$ is an example of how the $\widetilde{K}_c$ measure is an imperfect proxy for Komologrov complexity; both have higher complexity using alternative measures ($\widetilde{K}_s$=11 and $\widetilde{K}_{lz} \approx 211$).

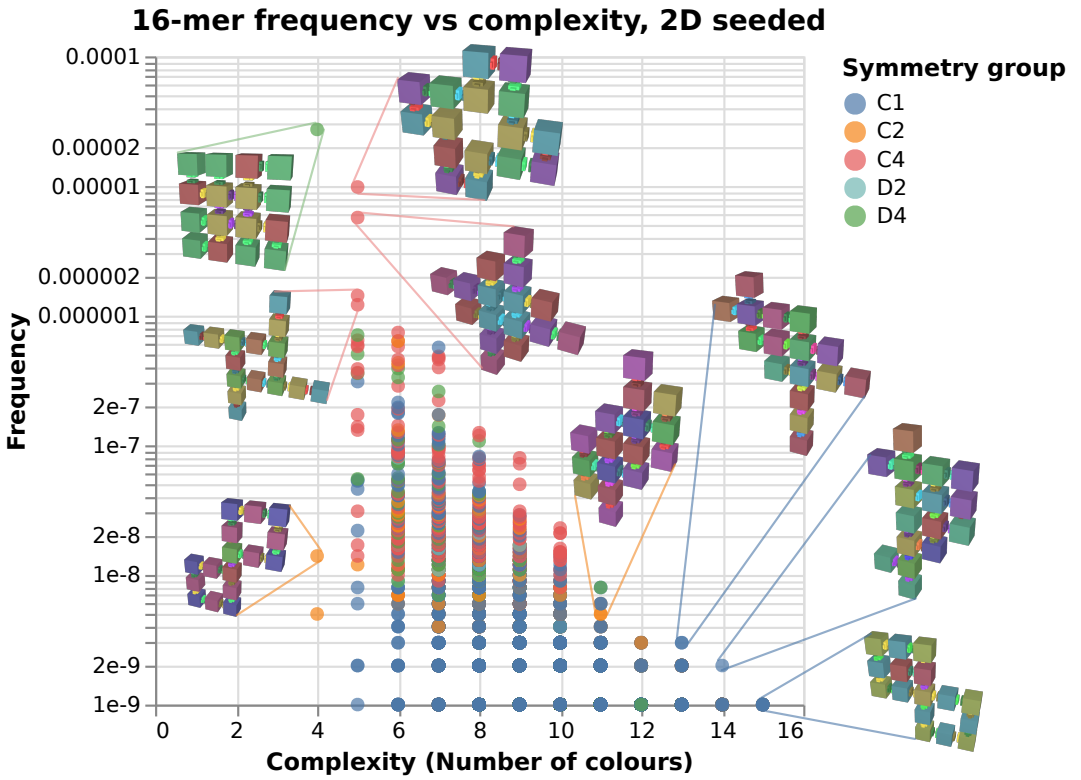


Figure 3.8: Frequency vs complexity ($\widetilde{K}_c$) of 16–mers found when sampling $I^{2D}_{16,31}$. Each point represents a unique polyomino shape, some of which are visualised.

3.4 Polycube samplings

Given the reference polyomino sampling's agreement with previous results, we now move on to also sample the three-dimensional polycube input space. These samplings compare results between 2D and 3D and the seeded versus unseeded assembly modes.

3.4.1 Input space analysis

All these samplings were done with 10^8 samples each in the $I_{5s,31c}$ space. The reason for only using five species per rule is to ensure enough valid output for stochastic 3D, which, as shown in Figure 3.9, is relatively low. In general, the seeded assembly mode is expected to generate more valid rules than when the seed is random. Different seeds might assemble into multiple different shapes, causing the rule to be deemed non-deterministic (while it might still be deemed deterministic with a fixed seed). Furthermore, with the third dimension, there are also more degrees of freedom, that is, more orientations and more patches per cube to attach to, leading to the significantly lower number of valid 3D rules seen in the figure.

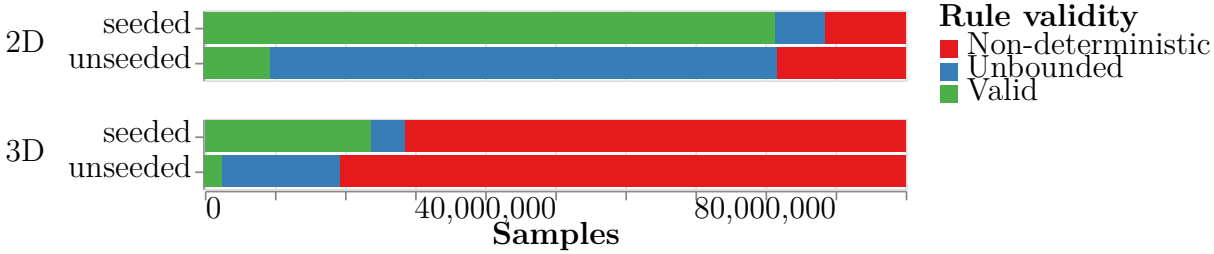


Figure 3.9: Proportion of valid rules when sampling $I_{5s,31c}$ for seeded and unseeded assembly in both 2D and 3D.

3.4.2 Output space analysis

Figure 3.10 shows the distribution of output sizes for the main samplings. As with the reference sampling, we can see that the number of shapes found initially is of the same order of magnitude as the total number of existing shapes for each size. Here we can also see even more clearly how shapes of certain sizes show up much more than their neighbours. As in the reference polyomino sampling, we can see

spikes for shapes with sizes that are multiples of four. For 3D, we can also see spikes for sizes 54 and 66, which are multiples of six rather than four, which makes sense as three-dimensional cubes have six patches each.

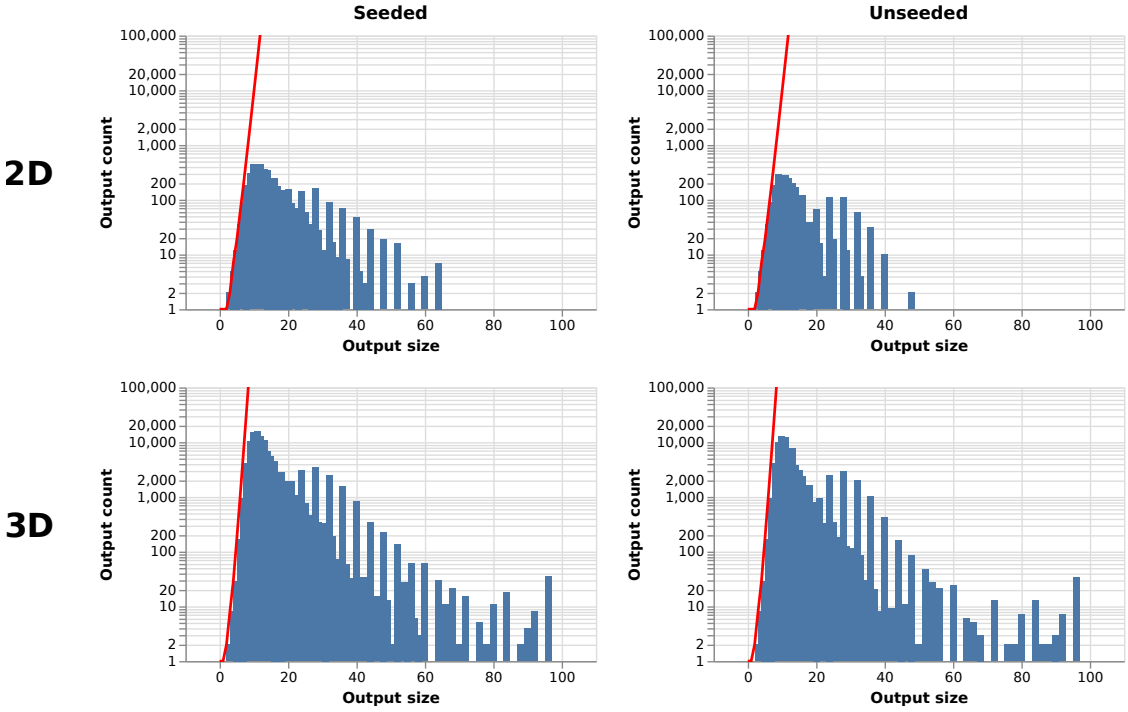


Figure 3.10: Distribution of output sizes when sampling $I_{5s,31c}$ for seeded and unseeded assembly in both 2D and 3D. The red line shows the actual number of polyominoes of each size, obtained from OEIS A000988 and A000162 [34, 35] respectively.

Figure 3.11 shows a gallery of the 8–mer polycubes found when sampling $I^{3D}_{5,31}$. Here, the difference in frequency between the shapes was so significant that they instead had to be scaled in proportion to the natural logarithm of their frequency in order to be visible in the same figure. Note that there are a total of 13,079,255 possible 16-mer polyominoes, while the total of 8-mer polycubes is 6,922 [34, 35], so we are now, compared to Figure 3.7, seeing a much larger portion of the output space and therefore more lower-frequency structures. Once again, we note how the most common shapes seem to be relatively simple, using a low number of species and colours.

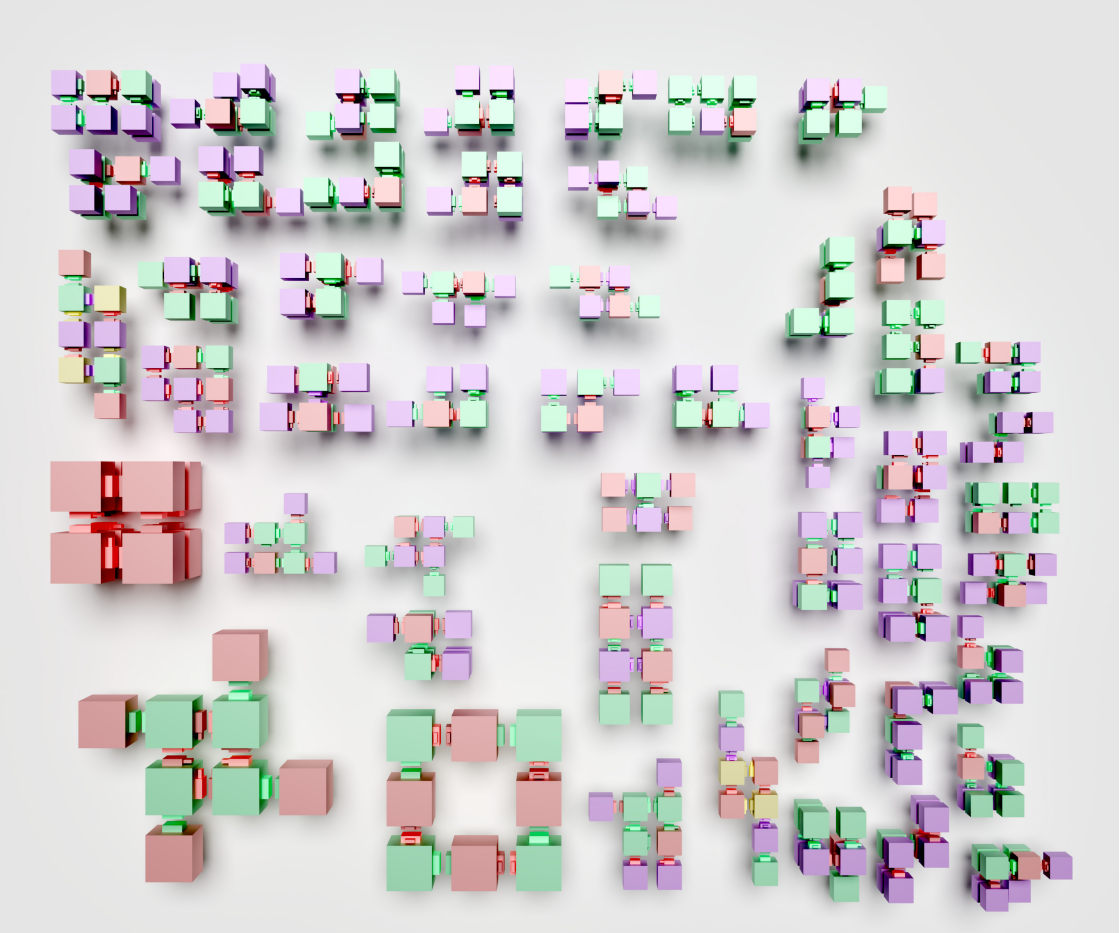


Figure 3.11: The 50 most common three-dimensional 8–mers, scaled proportional to the natural logarithm of their frequency.

3.4.3 Simplicity bias

As seen in Figure 3.12, the simplicity bias is still present for 3D polycubes, even using alternative measures of the complexity. The measures provide different resolutions, with the largest number of distinct complexity values for $\widetilde{K}_{lz}$, but all show that the frequency of a shape has an upper bound determined by its complexity.

One apparent break from the trend shown is the mostly constant $\widetilde{K}_s$ values for unseeded samplings (Figure 3.12.a and c); almost all the unseeded output shapes use the maximum number of species. This can be explained as a consequence of the determinism check. A seeded rule can include non-binding species (without any matching colours) that are simplified away and not counted, as long as none of them is the first cube. But with unseeded assembly, any species can be the

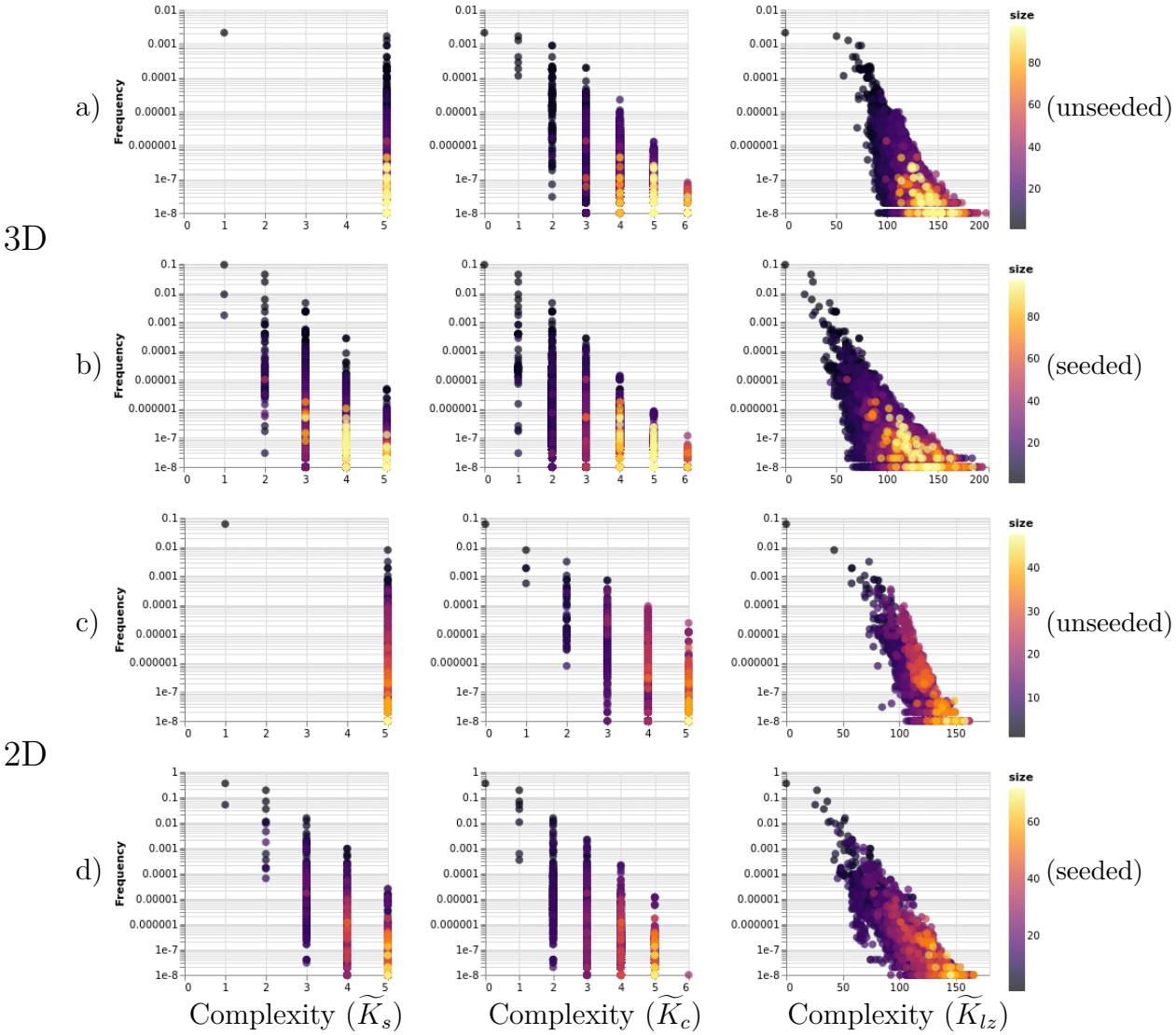


Figure 3.12: Frequency of shapes found when sampling $I_{5s,31c}$, versus different measures of their complexity. Each point is a unique polycube (or polyomino) shape, coloured by the size (number of cubes). Each column shows a different complexity measure, as proxies for their Komologrov complexity (See Section 2.3.4). The left column measures the minimum number of species required to assemble each shape, the middle instead shows the minimum number of colours required, and the right column is the shortest Lempel–Ziv-compressed rule. **a)** Unseeded assembly in 3D. **b)** Seeded assembly in 3D. **c)** Unseeded assembly in 2D. **d)** Seeded assembly in 2D. Note that the vertical axis (Frequency) is logarithmic.

seed, so when a non-binding species is used as seed the rule would be deemed non-deterministic and not included. A solution would be to remove the non-binding cubes from the unseeded rules, but then the simplification process would actually affect the assembly result (making non-deterministic rules deterministic), which

is something we want to avoid. Thus, for the unseeded results, we have to rely on the other two complexity measures.

3.5 Conclusion

The polycube model provides a simple way to self–assemble and design modular shapes. The stochastic assembler presented provides a method to quickly evaluate input candidates, ensuring that they give the intended output.

We have shown how large samples of the input space give valuable insight into the frequency of complex and simple shapes. Following AIT arguments, we expected low–complexity shapes to have a higher frequency, something we have now confirmed to also be the case, both when verifying the polycube model against previous 2D data and for the novel three–dimensional polycube mapping. This simplicity bias was shown to hold true for two different assembly modes and three different proxies for Komologrov complexity, as seen in 3.12.

4

Designing polycube assembly rules

Contents

In the previous chapter, we used random input rules to explore the properties of the corresponding distributions of output polycube shapes. However, the reverse problem is just as significant; given a target shape, how do you find a rule that assembles it?

This chapter provides a method for systematically enumerating valid solutions and discovering minimal solutions for arbitrary shapes. Section 4.1 describes how this is done, with the following sections providing examples of designed shapes and their assembly dynamics.

A trivial solution for designing a polycube shape would be to use *fully addressable assembly*: simply assign a unique species to each cube and a unique colour to each pair of adjacent patches. This is similar to the design principle underlying DNA bricks (Section 2.2.1), where every brick tile is unique. However, as was seen in Chapter 3, many shapes have alternative solutions requiring widely different numbers of unique components.

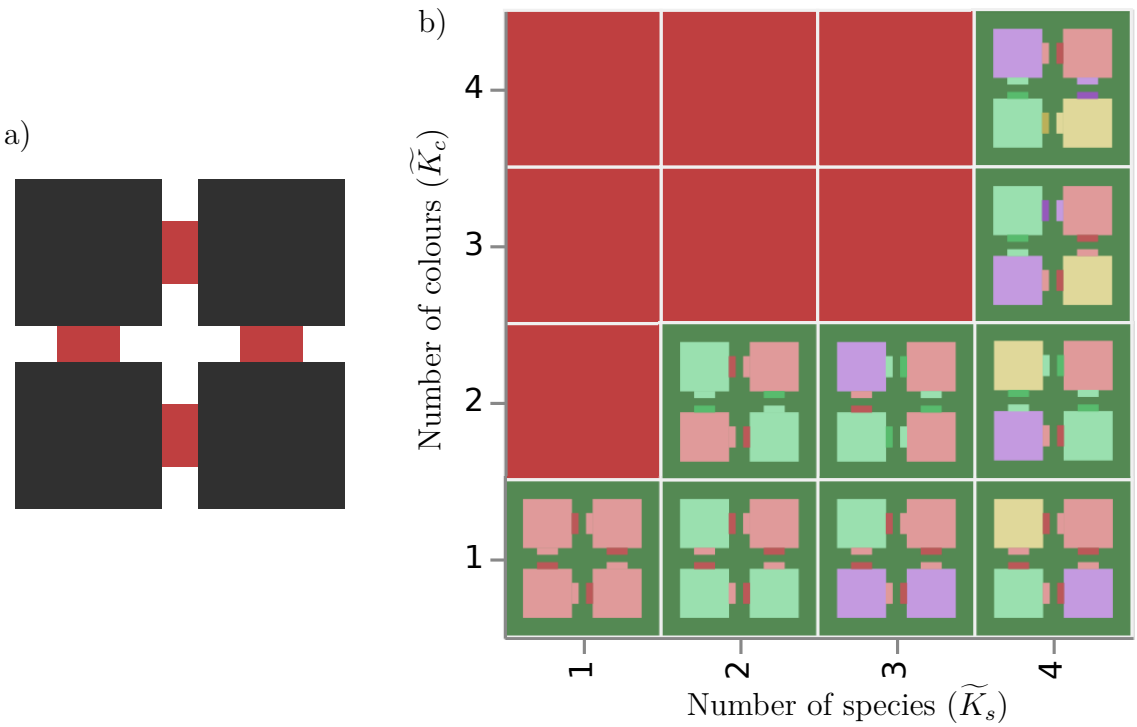


Figure 4.1: 2×2 square polyomino assembled with different levels of complexity. **a)** Schematic of the input shape, consisting of four connected tiles. **b)** The green region shows possible assembly solutions, from the *minimal solution* using a single species and a single colour (bottom left), to the *fully addressable solution* using four species and colours (top right). The red region lacks solutions.

Figure 4.1 shows a square tetromino with a variety of inputs assembling it, from the minimal solution with just a single species and one colour, up to the fully addressable solution with four species and four colours ($\widetilde{K}_s = \widetilde{K}_c = 4$). The

intermediate solutions are not necessarily deterministic in terms of which position gets which species, but they will always assemble into the same shape. Some solutions may also be able to form with only three bonds if the fourth patch pair has mismatched colours. However, such an assembly would be less stable than the assemblies shown in the figure.

It can be argued that the solutions on the bottom row, where $\widetilde{K_c} = 1$, all technically use only one species since all species have identical patches. However, one could imagine an additional property, such as different functionalisation, motivating the distinction. The empty red region in Figure 4.1.b) shows combinations of $\widetilde{K_s}$ and $\widetilde{K_c}$ for which a solution is not possible. For example, if you use a single species, you cannot use more than one colour.

So, how do we find alternative and simpler input rules for a shape? Surely, there must exist a better method than sampling the space of all rules (as done in Chapter 3)⁉ This chapter presents an approach where *satisfiability solving* is used to determine if a shape can be assembled from a given number of colours and species, thus automatically creating solution landscapes such as the one shown in Figure 4.1.

4.1 Satisfiability solving

Building upon a published method for determining patchy particle interactions for unbounded structures [36], it is possible to formulate and solve satisfiability problems that map onto assembly rules for the bounded polycubes.

In essence, we formulate a boolean expression that, if true, means it is possible to assemble a given polycube topology using a given number of colours and species. We can then use a satisfiability solver such as Glucose [37, 38] to check if that expression is indeed solvable and, if it is, extract an assembly rule from the solution.

4.1.1 Boolean expressions

The boolean expression is written in conjunctive normal form (CNF), where variables are composed into clauses using *NOT* ($\neg$) and *OR* ($\vee$) operators and where

the clauses are joined by *AND* ($\wedge$) operators. As a simple example, see the expression below:

$$(\neg x_{rain} \vee x_{umbrella} \vee \neg x_{walk}) \wedge (\neg x_{rainbow} \vee x_{rain}) \wedge (\neg x_{rainbow} \vee x_{sunny})$$

The first clause is $true$ for all values except when $x_{rain} = true$, $x_{umbrella} = false$ and $x_{walk} = true$; so the solution of taking a walk in the rain without an umbrella is forbidden. This could also be written as an *implication*: $x_{rain} \wedge x_{walk} \implies x_{umbrella}$.

The following two clauses in the example above are the CNF form of another implication: $x_{rainbow} \implies x_{rain} \wedge x_{sunshine}$, stating that a rainbow implies that we have both rain and sunshine (we cannot have a rainbow without rain or without sunshine). The full expression is satisfiable, for example, if we set $x_{sun} = true$, $x_{rain} = false$, $x_{walk} = false$, $x_{umbrella} = false$, and $x_{rainbow} = false$; ignoring the walk in the sunshine and remaining inside to work.

4.1.2 Polycube formulation

For the polycube problem, we introduce the following variables:

$x^A_{l,p,o}$ (patch p at position l has orientation o)

$x^B_{c_i,c_j}$ (colour c_i is compatible with colour c_j)

$x^C_{s,p,c}$ (patch p on species s has colour c)

$x^D_{p_1,o_1,p_2,o_2}$ (patch p_1 with orientation o_1 binds to patch p_2 with orientation o_2)

$x^F_{l,p,c}$ (patch p at position l has colour c)

$x^O_{s,p,o}$ (patch p on species s has orientation o)

$x^P_{l,s,r}$ (position l is occupied by species s rotated by r)]

	Clause	Boolean expression	Description
(i)	$C^{B}_{c_i,c_j,c_k}$	$\neg x^{B}_{c_i,c_j} \vee \neg x^{B}_{c_i,c_k}$	Each colour is compatible with *exactly one* colour.
(ii)	C^{C}_{s,p,c_k,c_l}	$\neg x^{C}_{s,p,c_k} \vee \neg x^{C}_{s,p,c_l}$	Each patch has *exactly one* colour.
(iii)	$C^{P}_{l,s_i,r_i,s_j,r_j}$	$\neg x^{P}_{l,s_i,r_i} \vee \neg x^{P}_{l,s_j,r_j}$	Each lattice position contains a single species with an assigned rotation.
(iv)	$C^{BF}_{l_i,p_i,c_i,l_j,p_j,c_j}$	$\left(x^{F}_{l_i,p_i,c_i} \wedge x^{F}_{l_j,p_j,c_j}\right) \Rightarrow x^{B}_{c_i,c_j}$	Adjacent patches in the lattice must have compatible colours.
(v)	$C^{rotC}_{l,s,r,p,c}$	$x^{P}_{l,s,r} \Rightarrow \left(x^{F}_{l,p,c} \Leftrightarrow x^{C}_{s,\phi_r(p),c}\right)$	Patches at a lattice position are coloured according to the (rotated) occupying species (see Table 4.2)
(vi)	C^{alls}_{s}	$\bigvee_{\forall l,r} x^{P}_{l,s,r}$	All species are required in the solution.
(vii)	C^{allc}_{c}	$\bigvee_{\forall s,p} x^{C}_{s,p,c}$	All patch colours are required in the solution.
(viii)	C^{O}_{s,p,o_k,o_l}	$\neg x^{O}_{s,p,o_k} \vee \neg x^{O}_{s,p,o_l}$	Each patch is assigned *exactly one* orientation.
(ix)	$C^{DA}_{l_i,p_i,c_i,l_j,p_j,c_j}$	$\left(x^{A}_{l_i,p_i,c_i} \wedge x^{A}_{l_j,p_j,c_j}\right) \Rightarrow x^{D}_{p_i,c_i,p_j,c_j}$	Adjacent patches in the target lattice must have the same orientation.
(x)	$C^{rotO}_{l,s,r,p,o}$	$x^{P}_{l,s,r} \Rightarrow \left(x^{A}_{l,p,o} \Leftrightarrow x^{O}_{s,\phi_r(p),o}\right)$	Patches at a lattice position are oriented according to the (rotated) occupying species.

Table 4.1: Polycube SAT clauses and their descriptions.

We then formulate clauses to constrain the problem, seen in Table 4.1. Clauses (i)-(vii) are the same as in [36] while the remaining are added, together with variables x^D, x^A and x^O above, to include *torsional restrictions*, meaning that patches need to bind at a compatible orientation (compared to being allowed to rotate freely).

Note that for 3D polycubes, there are six patches per species instead of the four seen in the 2D polyominoes. Compared to the 4 square rotations defined for 2D [36], this also introduces 27 possible cube rotations, seen in Table 4.2, where $\phi_r(p)$ corresponds to the patch number that will overlap with original patch number p at rotation r

Table 4.2: List of all possible rotations to superpose a cube onto itself. The value at index p in a mapping corresponds to the patch number that is now positioned where patch number p was before the rotation.

Rotation r	Mapping ϕ_r
0	(0, 1, 2, 3, 4, 5)
1	(0, 1, 3, 2, 5, 4)
2	(0, 1, 4, 5, 3, 2)
3	(0, 1, 5, 4, 2, 3)
4	(1, 0, 2, 3, 5, 4)
5	(1, 0, 3, 2, 4, 5)
6	(1, 0, 4, 5, 2, 3)
7	(1, 0, 5, 4, 3, 2)
8	(2, 3, 0, 1, 5, 4)
9	(2, 3, 1, 0, 4, 5)
10	(2, 3, 4, 5, 0, 1)
11	(2, 3, 5, 4, 1, 0)
12	(3, 2, 0, 1, 4, 5)
13	(3, 2, 1, 0, 5, 4)
14	(3, 2, 4, 5, 1, 0)
15	(3, 2, 5, 4, 0, 1)
16	(4, 5, 0, 1, 2, 3)
17	(4, 5, 1, 0, 3, 2)
18	(4, 5, 2, 3, 1, 0)
19	(4, 5, 3, 2, 0, 1)
20	(5, 4, 0, 1, 3, 2)
21	(5, 4, 1, 0, 2, 3)
22	(5, 4, 2, 3, 0, 1)
23	(5, 4, 3, 2, 1, 0)

4.1.3 On the importance of torsional interactions

It is possible to use the solver without any constraints on the patch orientations (as it was done in [36]). However, if we wanted to use the stochastic assembler from Chapter 3, orientations would have to be assigned randomly, resulting in a combinatoric explosion of additional assembly paths.

More importantly, the assembly should benefit from torsional patch interaction (for 2D polyominoes, this corresponds to the requirement that tiles can be rotated in the plane but not flipped). Figure 4.2 shows two versions of a simple rule, the only difference being the orientation of a single patch. While co–operative binding might, in such a case, benefit the desired square assembly, the self–limiting ability

would be significantly improved if the patches were torsionally rigid.

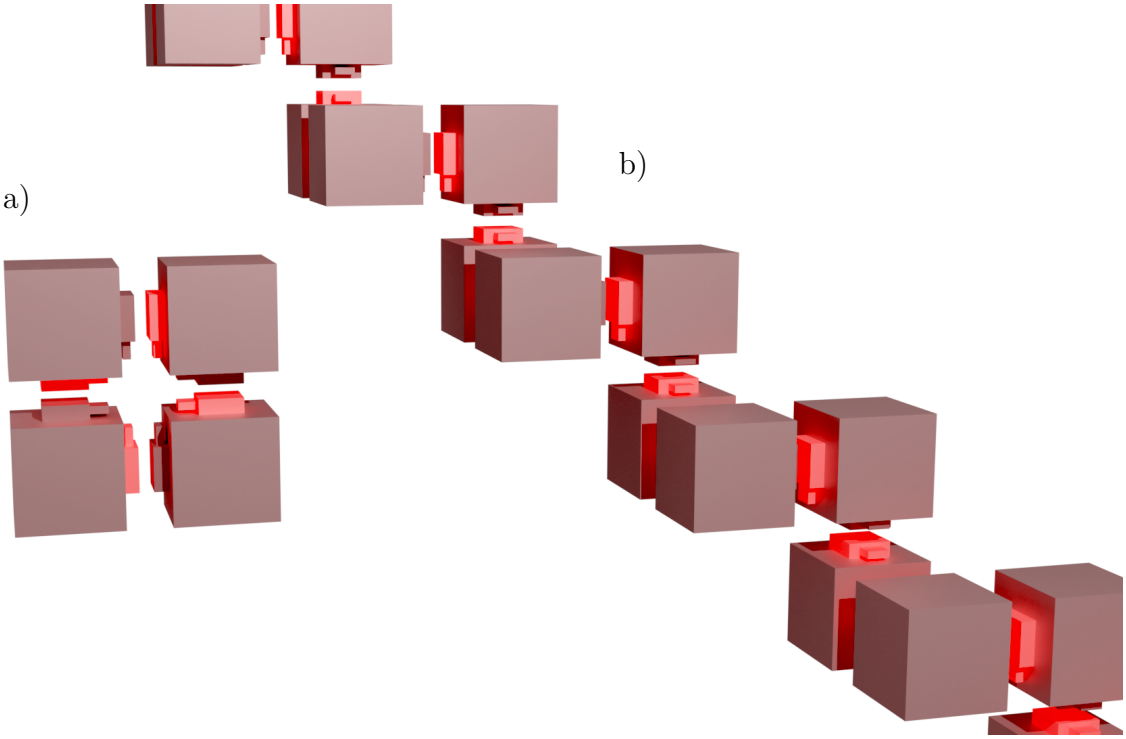


Figure 4.2: The consequences of a rotated patch. **a)** A minimal solution (one species and one colour) for a 2×2 square. **b)** The same rule as a), except one patch on is rotated by $\frac{\pi}{2}$.

4.1.4 Bounded structures

Besides the torsional patches, another important difference to [36] is that the method presented here allows for bounded structures. This is achieved by adding species of type *empty* as a "shell" around the shape to ensure that empty patches remain unbound. Adding a clause $x^B_{0,1}$ ensures that the colour 0 (on the shell) binds to colour 1 (on the boundary).

We then add clauses $x^F_{l,p,1}$ to constrain every boundary patch p at lattice position l to have the colour 1 and thereby not bind anything else. For example, in Figure 4.3, where these boundary patches are seen coloured white (and bordering an empty square), we get the following 12 clauses:

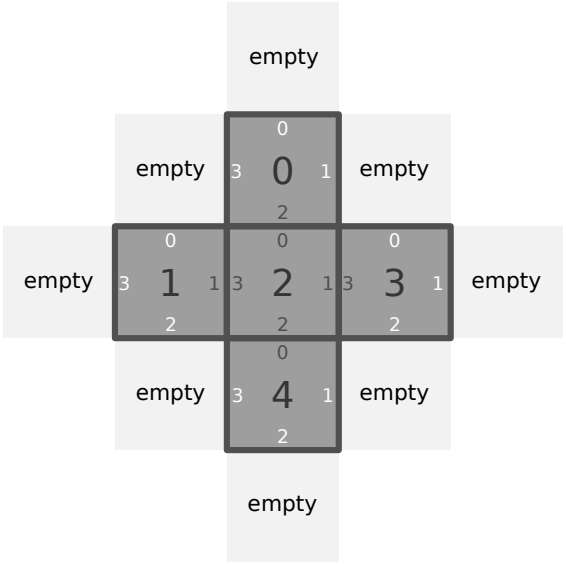


Figure 4.3: Bounded shape topology for satisfiability solving. Patches at the boundary of the shape (white) are constrained to only bind to "empty". 3D shapes are specified the same way, but with six patches per species.

$$
\begin{aligned}
&x^F_{0,0,1} \wedge x^F_{0,1,1} \wedge x^F_{0,3,1} \wedge \\
&x^F_{1,0,1} \wedge x^F_{1,2,1} \wedge x^F_{1,3,1} \wedge \\
&x^F_{3,0,1} \wedge x^F_{3,1,1} \wedge x^F_{3,2,1} \wedge \\
&x^F_{4,1,1} \wedge x^F_{4,2,1} \wedge x^F_{4,3,1}
\end{aligned}
\tag{4.1}
$$

The topology of the shape is enforced by clause (iv), $\left(\neg x^F_{l_1,p_1,c_1} \vee \neg x^F_{l_2,p_2,c_2} \vee x^B_{c_1,c_2}\right)$ in CNF, making sure that if patch p_1 on lattice position l_1 binds to p_2 on lattice position l_2, their colours are compatible. Similarly, clause (ix) ensures that the patches have the same orientation.

4.1.5 Interaction matrix

Compared to [36], the polycube interaction matrix is by default fixed. Thus, the $x^B_{c_i,c_j}$ variable has fixed values and we only need to extract the values of $x^F_{l,p,c}$ and $x^O_{s,p,o}$ to construct the assembly rule. Note, however, that it is still possible to re–enable a variable interaction matrix, something which could prove useful for some shapes where, for example, self–complementary patches would result in a lower complexity.

Compared to the interaction matrix convention used in Chapter 3, where each colour c binds to $-c$, the colour values in the SAT solver remain unsigned and

instead pair each even colour c to the odd $c + 1$. The colour pairs are mapped back to the polycube convention when obtaining the solution.

4.1.6 Assembly determinism

Even if the SAT solver determines that a solution exists, it is still possible that the rule we get can also assemble into other shapes. Recall, once more, the "giraffe duck" shape from Figure 3.2.b). If we solved for the shape with two neck cubes, the non–deterministic rule shown would be a perfectly valid solution according to the SAT solver, even though it can also produce giraffe ducks with any other neck length.

Because of this, we once again use the stochastic assembler to verify that the rule assembles into the correct shape every time. Each potential rule is evaluated a large number of times (by default 100), calculating an assembly ratio. If the ratio is 1, the rule is considered bounded and deterministic and, as such, a valid solution.

4.1.7 Finding the minimal assembly rule

By iteratively ruling out lower values of $\widetilde{K_s}$ and $\widetilde{K_c}$, a minimal solution can be found, as detailed in Figure 4.4. It is also possible to generate and compare alternative solutions of varying complexity. While forbidding an undefined solution and retrying could in principle, continue until the computer runs out of memory (or the solution is disproven or valid), the results presented below used a limit of 100 retries before moving on. It is thus possible, albeit unlikely, that the actual minimal solutions were missed, but the solutions found can at least be claimed to be minimised. Instead of interatively forbidding solutions, it is possible to use a solver like Relsat[1] to obtain multiple solutions at once, but this also has no guarantee of finding all solutions with the memory available.

The exploration of the solution landscape can be done in parallel, with each combination of $\widetilde{K_s}$ and $\widetilde{K_c}$ explored concurrently.

[1]Found at `https://code.google.com/archive/p/relsat`

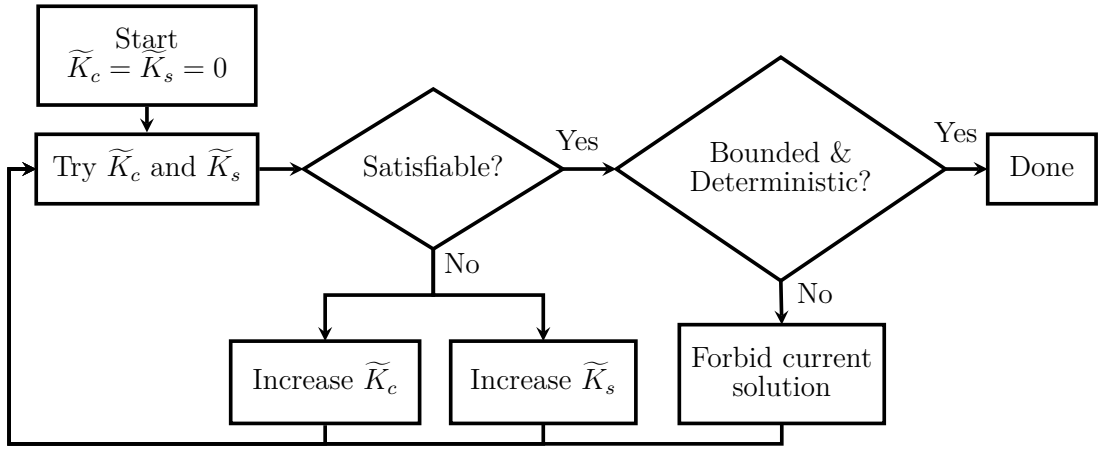


Figure 4.4: Algorithm for finding the minimal solution using SAT. Even if a solution is found to be satisfiable it might not assemble correctly every time. Additional solutions for a given $\widetilde{K}_c$ and $\widetilde{K}_s$ are found by explicitly forbidding the current solution.

4.2 Example solves

This section presents a set of different shapes solved to demonstrate the SAT solver method described above.

Swan A more sophisticated version of the "giraffe duck" from Figure 3.2, the Swan shape has a fixed neck length of one intermediate cube, as shown in Figure 4.5.a). Figure 4.5.b) shows the solution landscape, where valid deterministic solutions can be found along the upper diagonal. Many of the configurations that are classified as undefined (UND), because they are not fully deterministic, still assemble at a high ratio. This indicates that these $\widetilde{K}_s$, $\widetilde{K}_c$ combinations still could provide useful solutions.

Polyomino J With the 2D option enabled, the solver requires fewer clauses as it only needs to check four patches and four rotations per particle. Here this is used to explore the solution landscape for a polyomino shaped like a letter **J**, shown in Figure 4.6.a). Figure 4.6.b) shows the resulting solution landscape, where the minimal solution is just one species less than the fully addressable one. This can be explained by a lack of symmetry and modularity in the shape, where only species used for the "endpoints" are reusable.

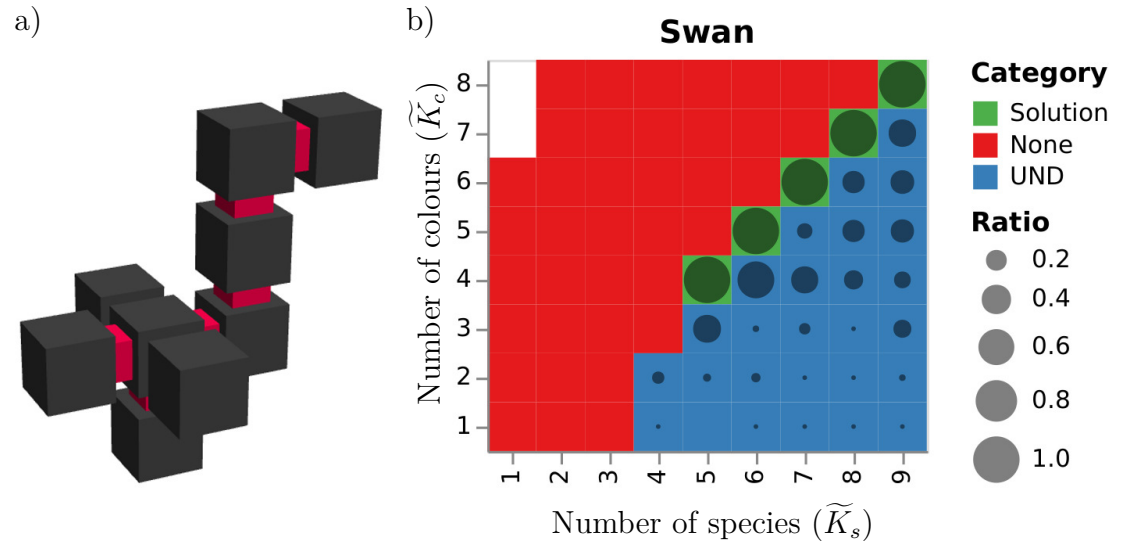


Figure 4.5: Designing a polycube "Swan". **a)** Visualisation of the swan shape. **b)** Solution landscape.

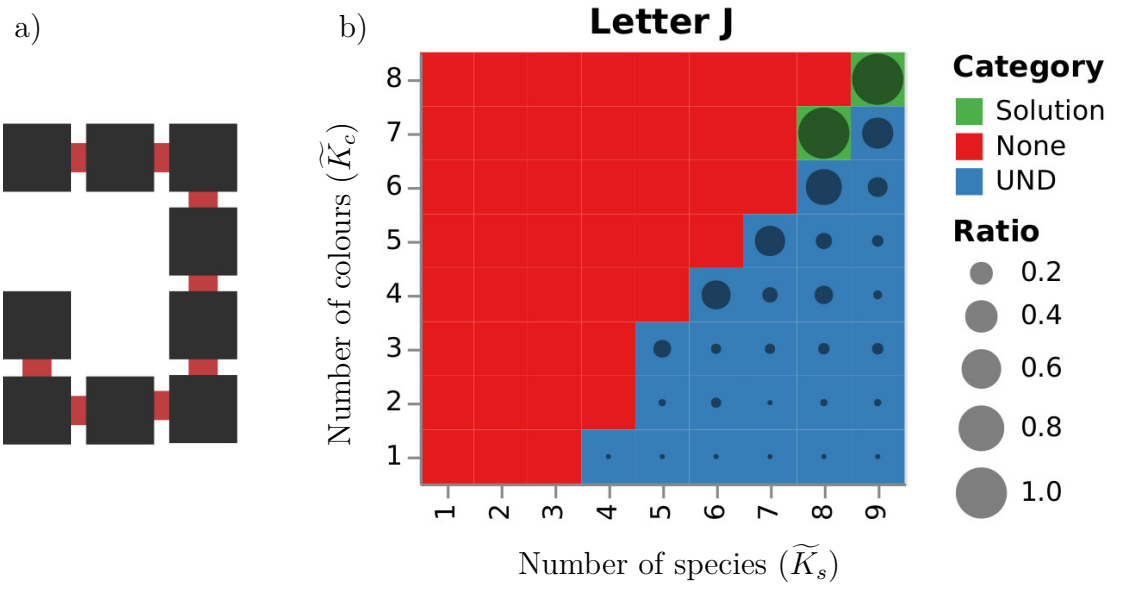


Figure 4.6: Designing a polyomino "letter J". **a)** Visualisation of the shape. **b)** Solution landscape.

Robot The "robot" shape seen in Figure 4.7.a) consists of more cubes than the previous examples, thereby resulting in the larger assembly landscape seen in Figure 4.7.b). Once again, the valid solutions clearly follow the border region between the blue UND region of solutions with varying assembly ratio and the red region where no solution is possible.

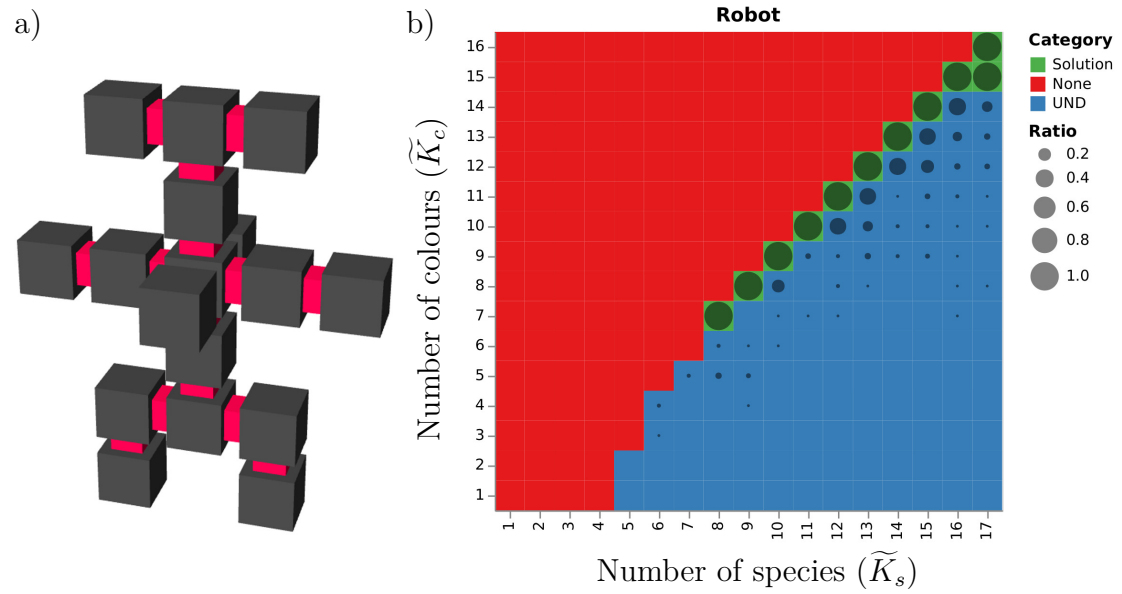


Figure 4.7: Designing a polycube "robot". **a)** Visualisation of the robot shape. **b)** Solution landscape.

Hollow cube The hollow $3 \times 3 \times 3$ cube (Figure 4.8.a) is a good example of a larger structure (20 cubes) that still has a very low complexity solution, as seen in Figure 4.8,b). This low complexity can be explained by the high symmetry of the shape, needing just a single species for the vertices and another species for the edges.

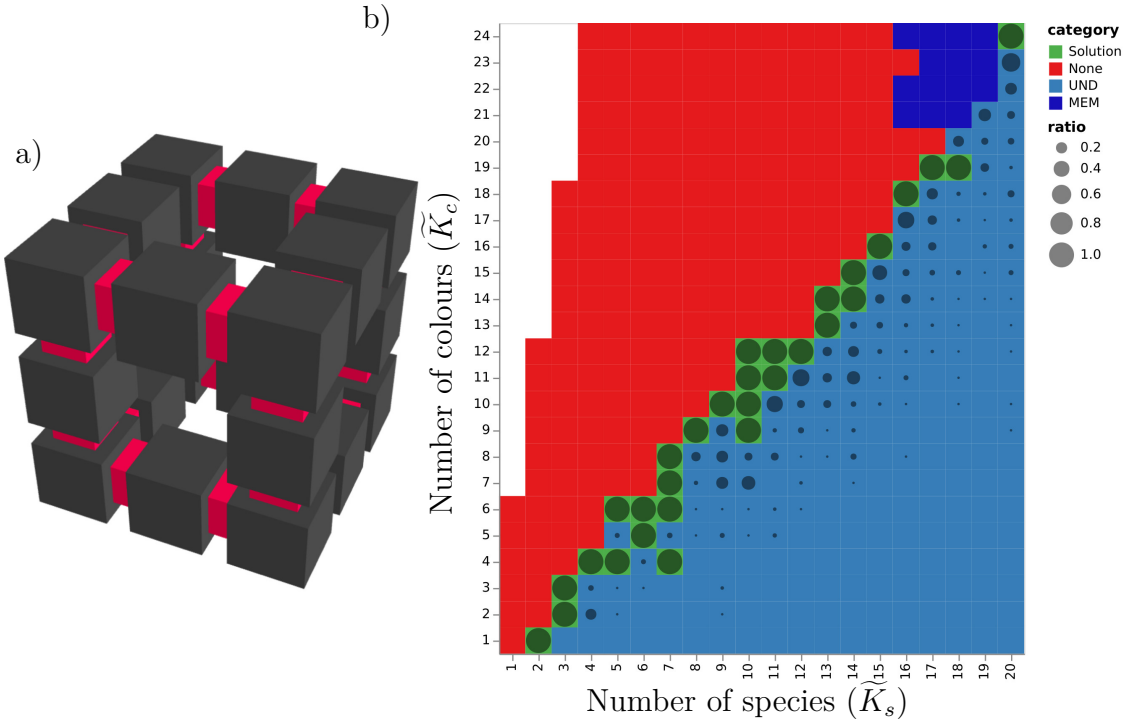


Figure 4.8: Designing a hollow $3 \times 3 \times 3$ cube. **a)** Visualisation of the $3 \times 3 \times 3$ cube shape. **b)** Solution landscape.

4.3 Scalability analysis

To find out how the method scales with the size of the system, we also solve solid, fully connected cube shapes of sizes 2, 3, and 4.

Size 2 cube The $2 \times 2 \times 2$ cube solutions, seen in Figure 4.9, highlights an important point with the current design pipeline. The stochastic assembly model verifies that the correct shape is assembled 100% of the time for the minimal solution using $\widetilde{K}_s = 2$ species and $\widetilde{K}_c = 1$ colour. However, inspecting the assembly (040087000400870087008700), it is clear that all patch pairs are not always connected. Since only the coordinates are compared, assemblies with mismatches will still be accepted. This approach was chosen in order to use the same polycube model as in Chapter 3. If needed, it should be possible to modify the model to also verify the assembly ratio of a given topology network (or to report mismatches, as is done in the JavaScript code), but binding cooperativity should provide a bias against the mismatched alternate assemblies.

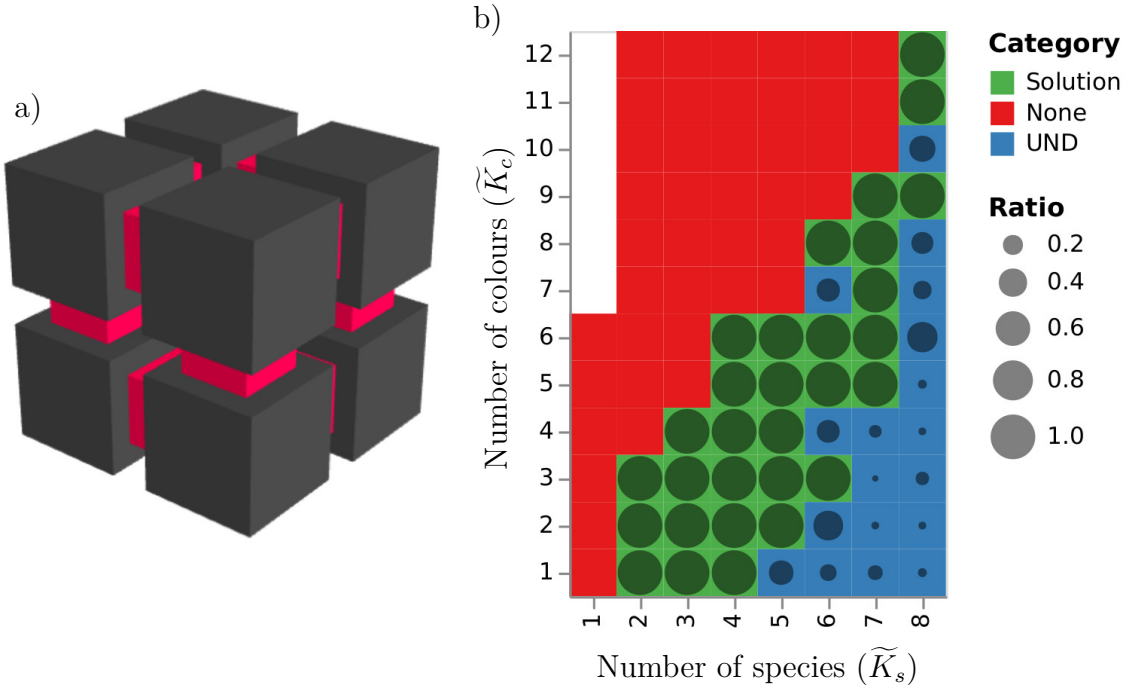


Figure 4.9: Designing a $2 \times 2 \times 2$ cube. **a)** Visualisation of the $2 \times 2 \times 2$ cube shape. **b)** Solution landscape.

Size 3 cube Figure 4.10 shows the solution landscape for a solid $3 \times 3 \times 3$ cube. While it is visible also for the hollow cube, the sharp border of valid solutions along the diagonal seen in previous solutions is now less clear. This could be due to an increasing number of alternative solutions available for a given $\widetilde{K}_s$, $\widetilde{K}_c$ position, leading to some positions getting classified as UND while a valid solution could be found through further retries.

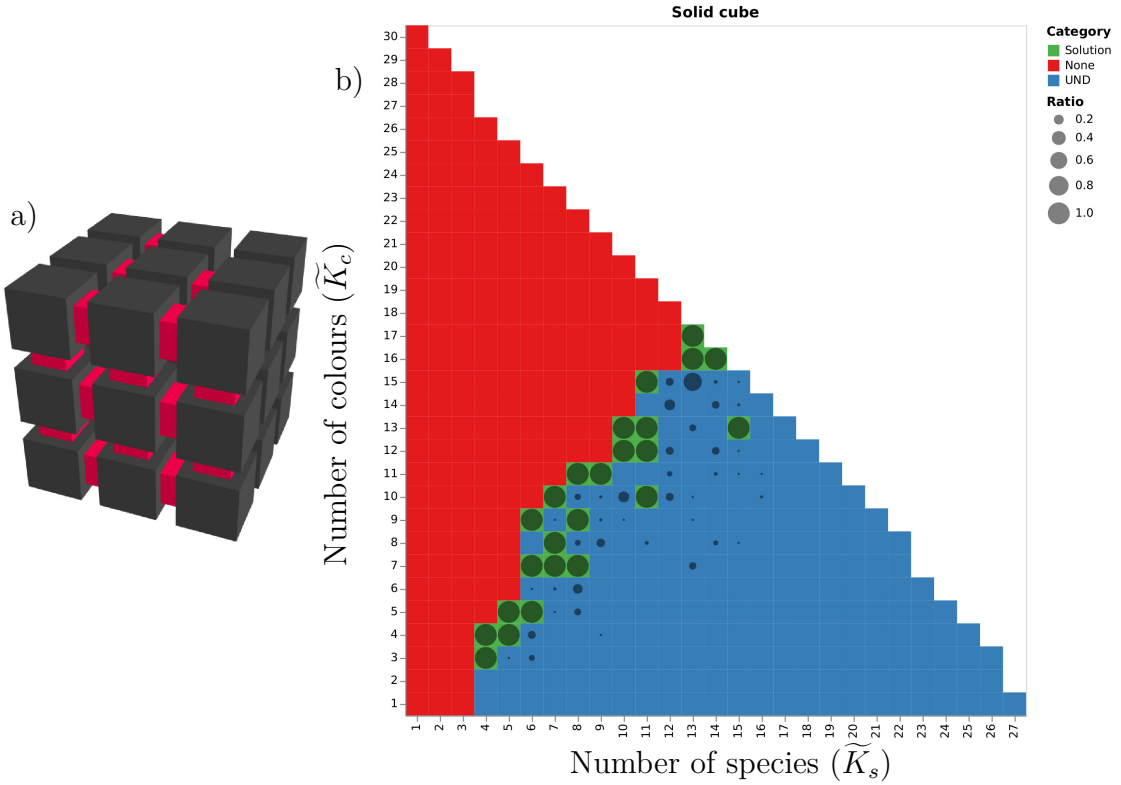


Figure 4.10: Designing a solid $3 \times 3 \times 3$ cube. **a)** Visualisation of the $3 \times 3 \times 3$ cube shape. **b)** Solution landscape.

Size 4 cube Figure 4.11 shows the solution landscape for a solid $4 \times 4 \times 4$ cube. The full solution landscape for this cube size is cropped to limit the number of needed solves and make the figure more readable. Similar to the solid $2 \times 2 \times 2$ cube, the minimal solution found here deterministically assembles the correct shape, but sometimes has mismatched connections.

An interesting comment for this shape is that a solid $4 \times 4 \times 4$ cube was found during the sampling in Chapter 3, which used only two species and one colour (840a070808070008080000 89). However, such a solution would however

never be suggested by the SAT solver, since it in no instance can assemble without mismatches (and has a different topology).

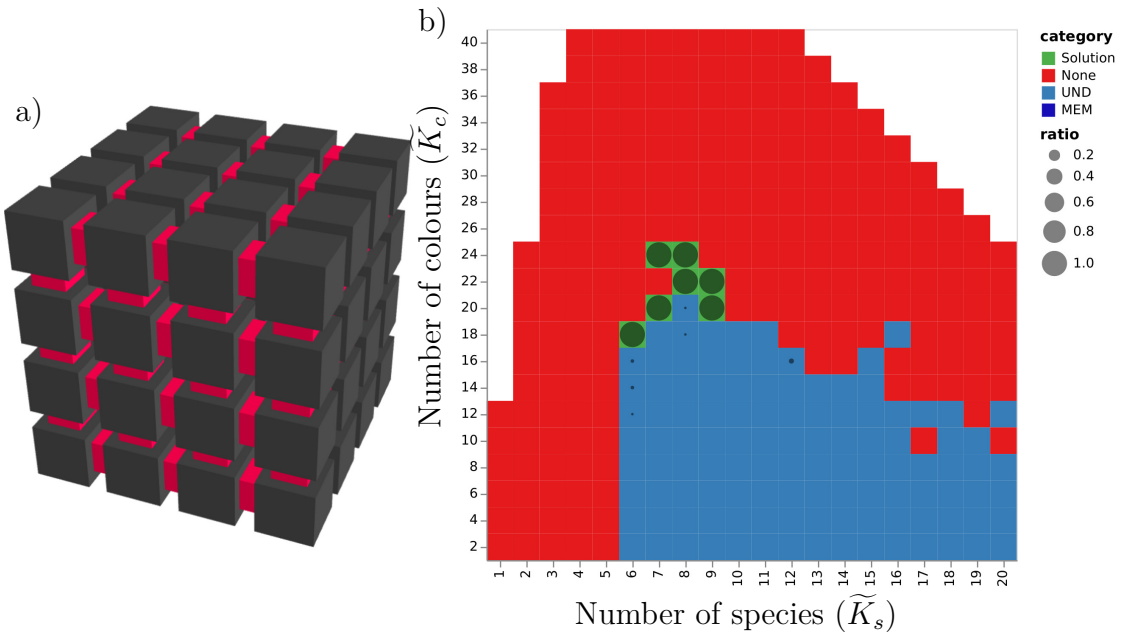


Figure 4.11: Designing a solid $4 \times 4 \times 4$ cube. **a)** Visualisation of the $4 \times 4 \times 4$ cube shape. **b)** Solution landscape.

Scaling of SAT clauses and variables Figure 4.12 shows how the number of variables and clauses in the SAT problem formulations increased with the size of the shapes. The number of required species and colours are much smaller for the minimal solutions, decreasing the problem size significantly, but the minimal solutions still require ever-larger problem formulations as the shape size increases.

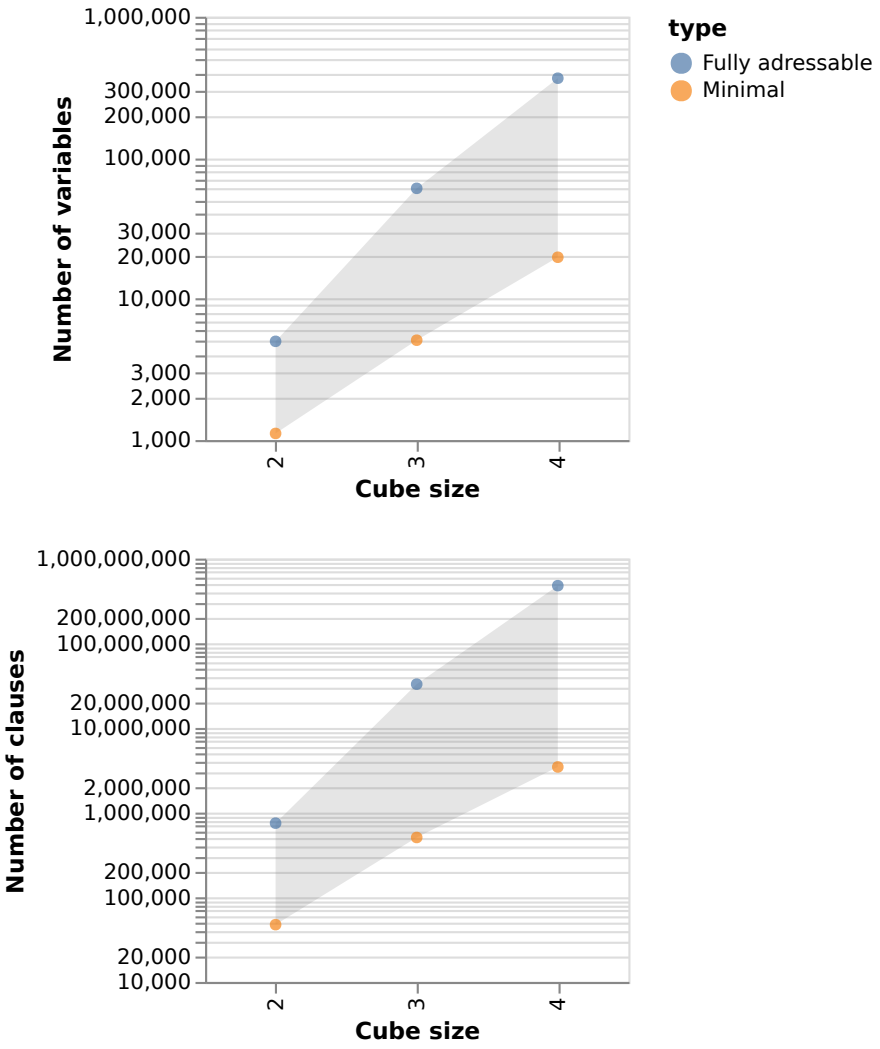


Figure 4.12: Scaling of the required number of SAT variables (top) and clauses (bottom) required in the problem formulation for the different solid cube sizes. The upper boundary (blue) shows the fully addressable solutions, while the lower boundary (orange) shows the numbers for the minimal valid solution. Note that the y-axes are logarithmic.

4.4 Comparison to random sampling results

In Chapter 3 we sampled the space of polycube input rules and looked at the resulting distribution of output complexities. Using the SAT solver approach presented in this chapter, we can now also sample the output shapes directly.

4.4.1 All octominoes

Using the SAT solver, we calculate the solution landscapes for all 369 possible 8–mer polyominoes. Figure 4.13 shows all of them ordered by the minimum number of

species found necessary for assembly. Note how most of the 8–mers require around six species, while some need eight (one species per position) and others only need two.

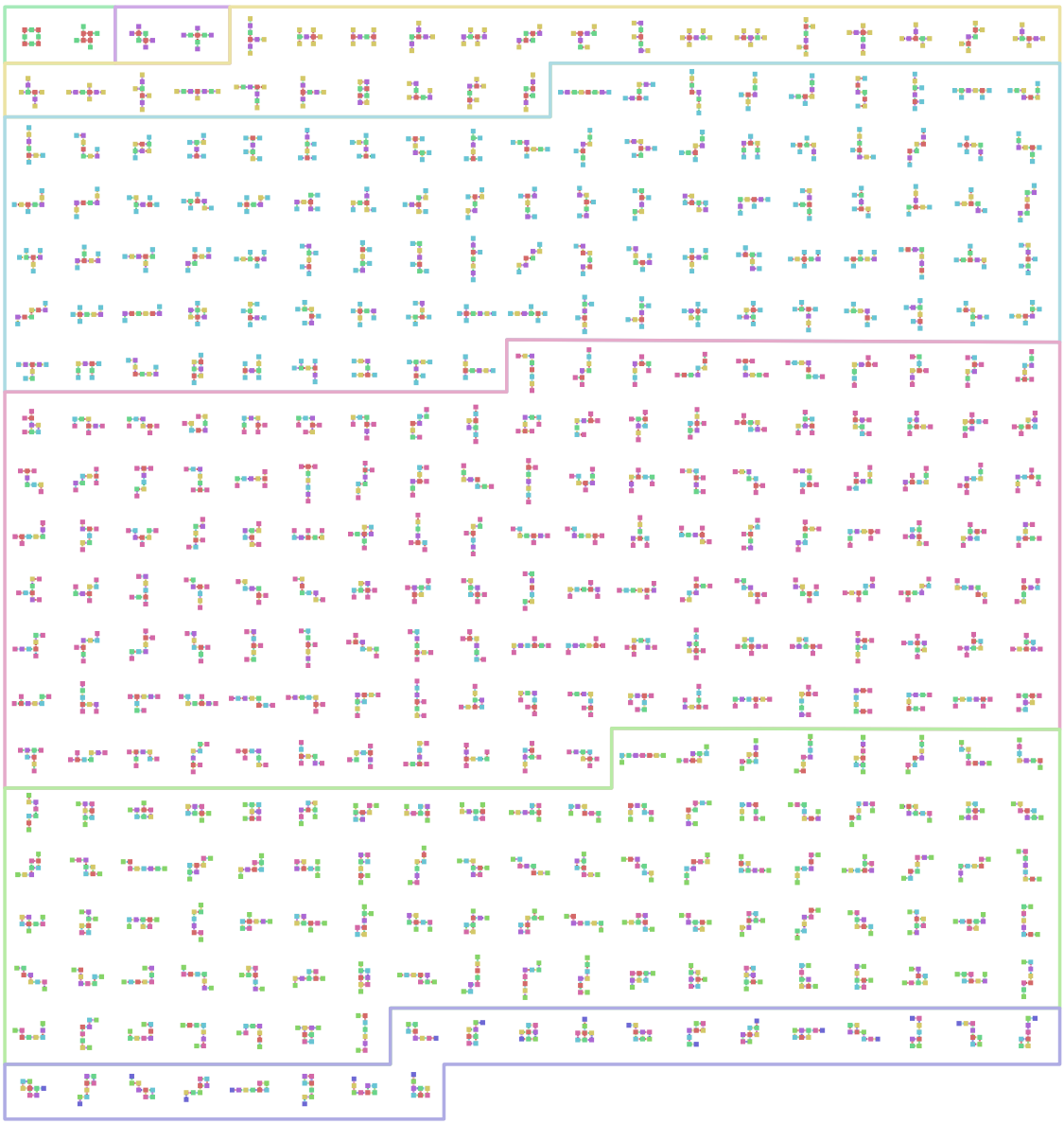


Figure 4.13: All 369 free 8–mer polyominoes (2D) grouped by their smallest solved complexity ($\widetilde{K}_s$). The polyominoes are ordered from left to right, top to bottom, primarily according to the minimal number of species ($\widetilde{K}_s$), secondarly the number of colours ($\widetilde{K}_c$) needed for their assembly. Lines are drawn to group the polyominoes with equal $\widetilde{K}_s$, showing that there are 2 polyominoes with $\widetilde{K}_s = 2$, 2 polyominoes with $\widetilde{K}_s = 3$, 25 polyominoes with $\widetilde{K}_s = 4$, 94 polyominoes with $\widetilde{K}_s = 5$, 135 polyominoes with $\widetilde{K}_s = 6$, 91 polyominoes with $\widetilde{K}_s = 7$, and 20 polyominoes with $\widetilde{K}_s = 8$.

This is more clearly visible in Figure 4.14, where we can see the complexity distribution of the solved 8–mer shapes compared to the 8–mer polyominoes found

when sampling $I^{2D}_{16_s,31_c}$ in Chapter 3. It is clear that the solved shapes generally use much fewer species and fewer colours compared to the shapes found with random sampling. Most sampled results even use more than eight species, which is more than any 8–mer shape should ever require.

This species redundancy is mainly an artefact of how the sampling was performed. The sampled $I^{2D}_{16_s,31_c}$ input space uses 16 species by default (so that it would be able to find any 16–mer, see Section 3.3), only removing species that could never bind. Additional species never present in the final assembly (for a given seed) were too computationally expensive to remove in the simplification step. An unseeded sample of $I^{2D}_{8_s,10_c}$ should give better 8–mer results. If finding minimal input rules is the goal, a non–uniform sampling biased toward fewer species and colours should also be more successful.

4.4.2 All hexacubes

As a three–dimensional example, let us also look at the 112 free 6–mer polycubes. Figure 4.15 shows all of them ordered by the minimum number of species found necessary for assembly. Like for the 8-mer polyominoes, we see that few require the fully addressable amount of species. A few can assemble with only two species, while the majority has a complexity somewhere in between.

We can see the normal distribution of solved species counts in Figure 4.16, also showing how the distribution of species and colours compare to those of the 6–mer polycubes found when sampling $I^{3D}_{5_s,31_c}$ in Chapter 3. Since the sampled input space was limited to a maximum of five species, we do not see the redundant species that were seen for the polyominoes in Figure 4.14.

Figure 4.16 also highlights another limitation with the sampled results; the number of 6-mer polycubes found, 166, is larger than the unique number of possible 6-mer polycubes (112). Likely, this is due to some rules, despite extensive testing, not being fully deterministic, thereby creating redundant polycube groups during the sampling. Since the polycube count is still within the correct order of magnitudes (Figure 3.10), adressing this discrepancy is deemed out of scope for this work.

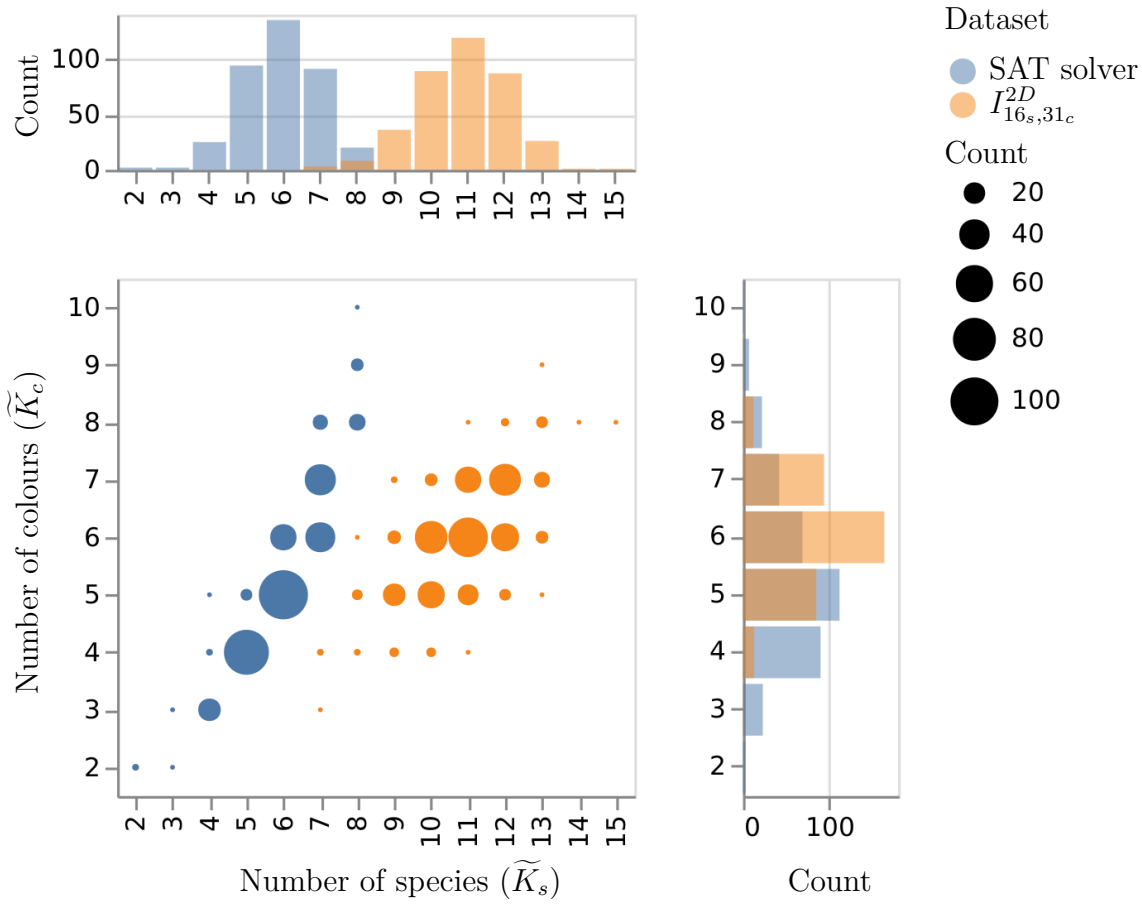


Figure 4.14: Complexity distributions for 8–mer polyominoes. The blue distribution (left) corresponds to the minimal SAT solver solutions to all 369 possible 2D 8–mers. The orange distribution (right) corresponds to the 8–mers found through randomly sampling the $I^{2D}_{16_s,31_c}$ space of polyomino rules.

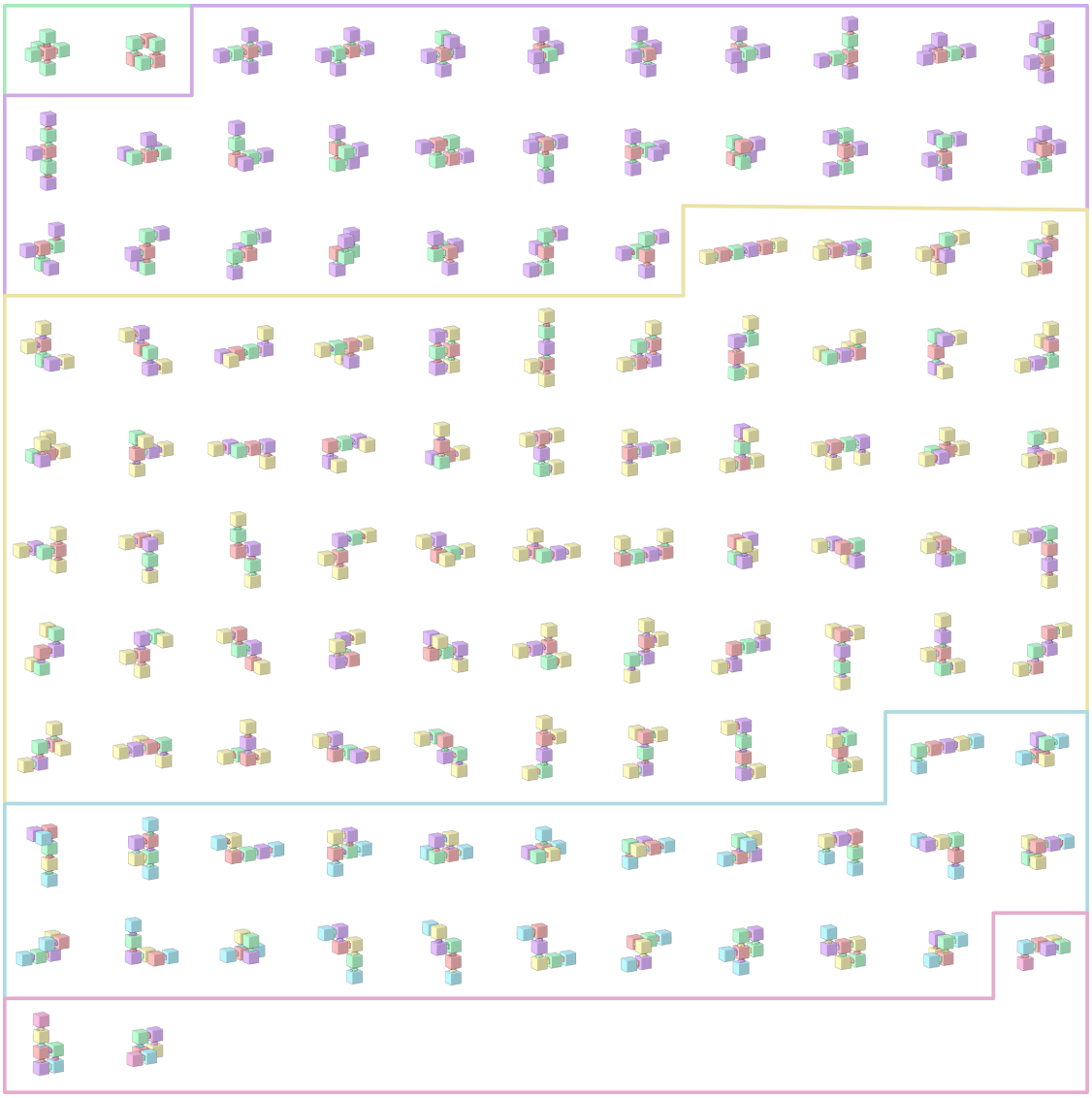


Figure 4.15: All 112 free 6–mer polycubes (3D) grouped by their smallest solved complexity ($\widetilde{K}_s$). The polyominoes are ordered from left to right, top to bottom, primarily according to the minimal number of species ($\widetilde{K}_s$), secondarly the number of colours ($\widetilde{K}_c$) needed for their assembly. Lines are drawn to group the polyominoes with equal $\widetilde{K}_s$, showing that there are 2 polycubes with $\widetilde{K}_s = 2$, 27 polycubes with $\widetilde{K}_s = 3$, 57 polycubes with $\widetilde{K}_s = 4$, 23 polycubes with $\widetilde{K}_s = 5$, and 3 polycubes with $\widetilde{K}_s = 6$

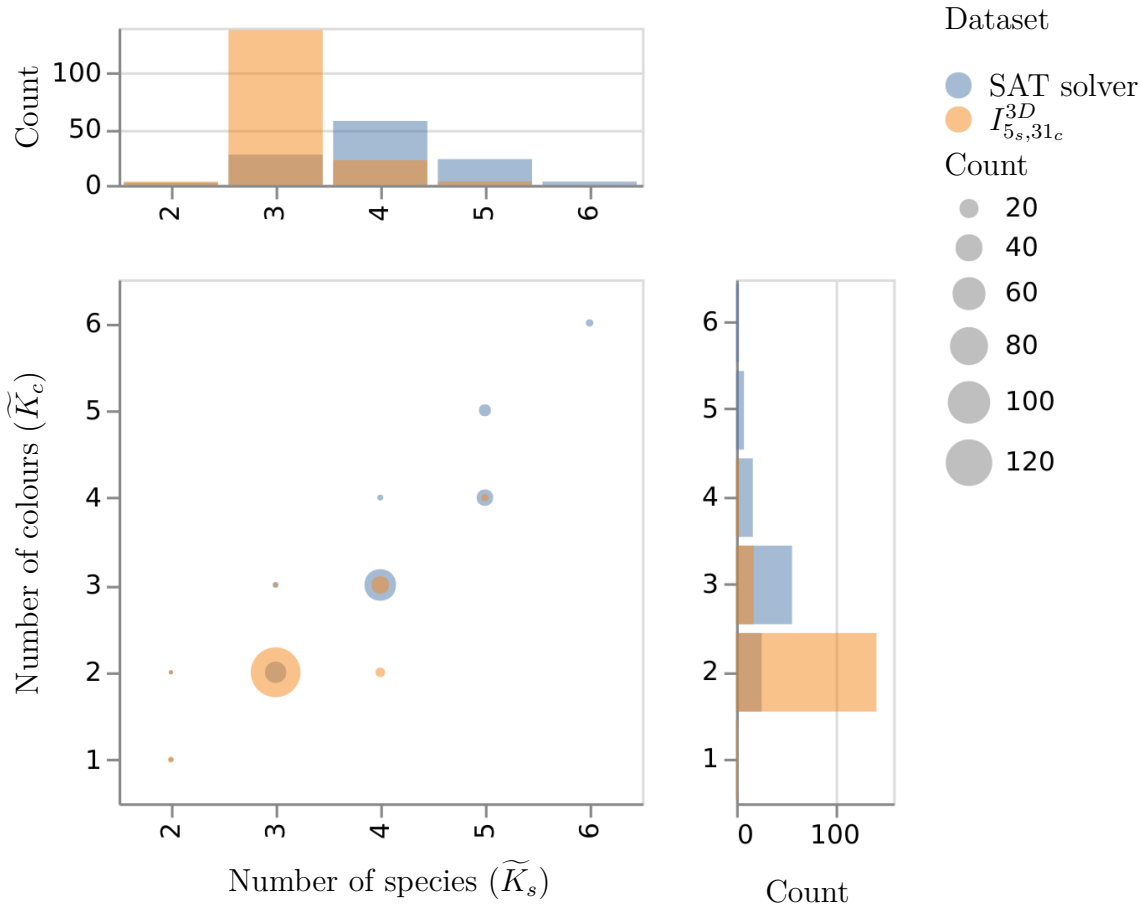


Figure 4.16: Complexity distributions for 6–mer polycubes. The blue distribution (left) corresponds to the minimal SAT solver solutions to all 112 possible 3D 6–mers. The orange distribution (right) corresponds to the 6–mers found through randomly sampling the $I^{3D}_{5_s,31_c}$ space of polycube rules.

4.5 Assembly in a continuous model

While the stochastic assembler works well to test determinism, it does not have realistic assembly dynamics. To study how the rule affects assembly dynamics, we now turn to a more sophisticated patchy particle simulation.

4.5.1 Patchy particle simulation

Besides discrete tile models, self–assembly can also be modelled using simulations of rigid–body spheres called *patchy particles*. The particles move using Molecular Dynamics (MD), updating their positions and orientations according to Newtown's laws of motion. An Andersen–like thermostat [39] is used to manage the energy in the system (there is no explicit solvent) and to keep a constant, specified temperature. Furthermore, each particle has one or more patches that can interact with and bind the patches of other particles. The angles at which two patches interact can be modulated to allow for wider or more narrow patches, allowing us to study how the bond flexibility affects the assembly dynamics. Further details of the patchy particle model are provided in Appendix B.

As seen in Figure 4.17, the patchy particle simulator included in the oxDNA package [40] has previously been used by Romano et al. to verify the patchy interactions designed by their SAT–solver method [36]. A derivative of that patchy particle model is used here, modified to include torsional interactions to account for the polycube requirement of patch orientation alignment.

4.5.2 Yield calculation

The patchy particle simulation yields are mainly calculated using edge-induced subgraph isomorphism [41]. We annotate $\sigma(G_a, G_b) ==$ True if the graph G_a is an edge-induced subgraph of the graph G_b. The connectivity graph G_i for each assembled particle cluster is compared to the graph for the intended shape $G_{correct}$. If G_i is a large enough edge-induced subgraph of G_{correct}, meaning that its particles are connected like a large enough subset (the current results use a

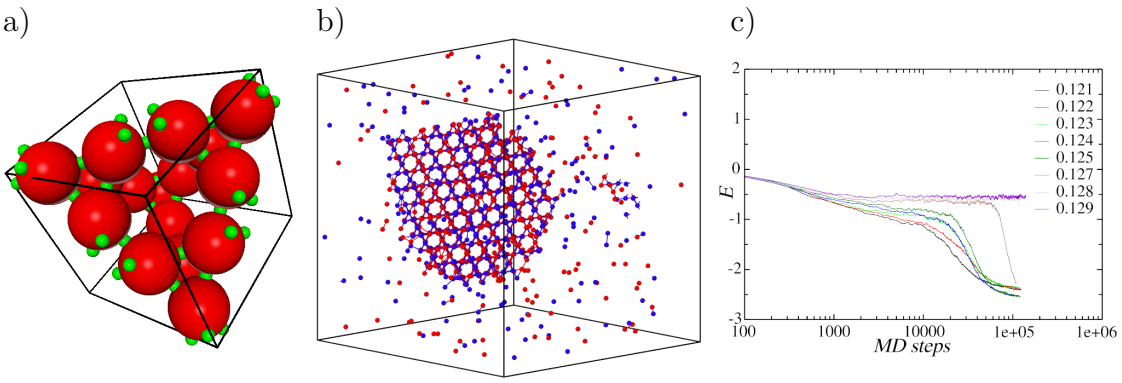


Figure 4.17: Patchy particle simulation, adapted from [36]. **a)** The unit cell of a tetrastack lattice build with patchy particles. **b)** Simulation snapshot of a forming tetrastack lattice. Note the free–flowing particles that have not yet attached the growing latttice they surround. **c)** Tetrastack particle energy plotted over simulation time for different temperatures. Sudden drops in energy correspond to nucleation events (where the lattices start forming).

75% size cutoff) of the correct shape, it contributes to the yield with its fraction of correctly assembled particles:

$$Y_c = \sum_{G_i \in c} \begin{cases} \frac{|N(G_i)|}{|N(G_{\text{correct}})|} & \text{if } \sigma(G_i, G_{\text{correct}}) \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The subgraph calculation becomes very compute-intensive for larger and more highly connected graphs, so for the $4 \times 4 \times 4$ solid cube we instead check the cluster's species composition, making sure that its species are a subset (with duplicates) of the intended shape. For the minimal solution, this underestimates the yield for some temperatures compared to the subgraph measure, as the solution can produce correct shapes with varying species compositions (but some mismatched connections).

The other exception is used in the case of the werewolf design (Section 4.6), where the connectivity graph of the human structure is a subgraph of the connectivity graph of the wolf structure. To tell the two shapes apart, we check both the connectivity graph and the species composition.

4.5.3 Simulation results

Next, we simulate and compare the assemblies of the fully addressable and minimal solutions to the shapes introduced in Section 4.2. Figure 4.18 shows the assembly yields for three different shapes, using either their minimal or their fully addressable

solutions. At low-enough temperatures, all solutions assemble with good yields. As can be seen, the minimal solutions perform just as well as the fully addressable. This is not very significant for the Letter J shape since the complexity difference is small (Figure 4.6). However, for the robot and swan, having similar assembly kinetics using 8 unique species instead of 17 (for the robot, Figure 4.7), or even 5 instead of 9 (for the swan, Figure 4.5) can provide considerable experimental resource savings.

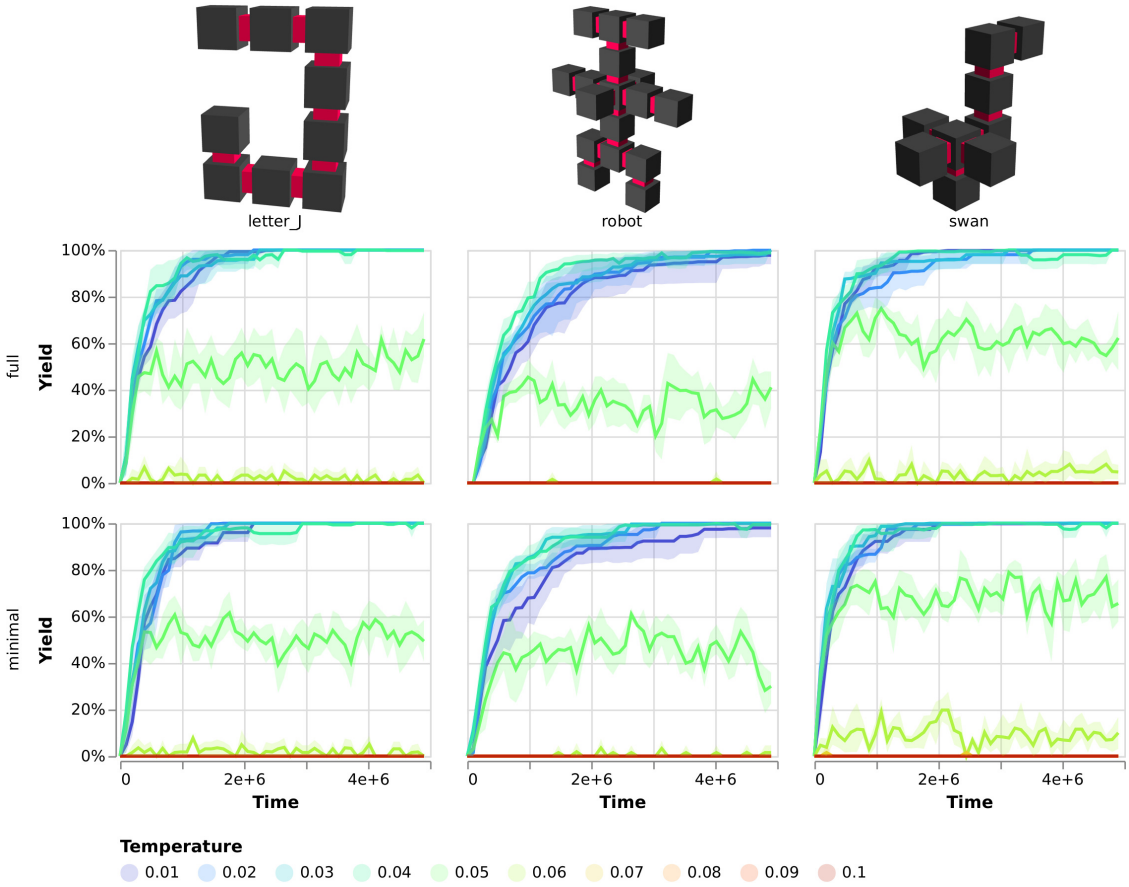


Figure 4.18: Assembly kinetics for three shapes. The top row column shows the fully addressable solution while the bottom shows the minimal solution. Solid lines are mean values from 5 duplicate simulations, error bands show the 95% confidence interval band. Each simulation is done using the narrow type 0 potential (patch width = 2.346) at a 0.1 particle density. Temperature and time is measured in simulation units.

Figure 4.19 shows the yield plots for three different solutions for assembling the hollow $3 \times 3 \times 3$ cube from Figure 4.8; the fully addressable solution with 20 species, a significantly smaller solution using four species, and the minimal solution with only two species. We also investigate the effects of bond flexibility by simulating

the assemblies at different patch interaction widths (defined in Figure B.1). More narrow patch widths (less flexible bonds) lead to slower assembly times, which can be expected since particles are less likely to bind. On the other hand, bonds that are too flexible can cause unwanted bonding and aggregation.

While the previous three shapes are unlikely to assemble into anything other than the intended size, the cubes can aggregate into clusters many times larger than intended if their bonds are flexible enough. This happens because cubes are self–limiting through their geometry, which works less well on flexible patchy particles compared to the rigid lattice of the stochastic assembler. In contrast, the previous shapes are self–limiting through their topology, with no loops in their connectivity graphs, so while flexible bonds can lead to deformations, their topology will remain the same. For patches without torsion, only the fully addressable solution manages to assemble with a good yield. More distinct species create larger connectivity graph loops, with a mitigating effect on aggregation with the cost of a longer assembly time.

The minimal solution, however, performs very well for the intermediate patch widths 0.955 and 0.657, assembling fast and at a high yield. These results show that not only can the minimised assembly sets provide resource savings, but they can also lead to faster assembly dynamics and better yields. For these designs, the patches are narrow enough to avoid aggregation, while the low species count improves the assembly speed by being more available (with one in two being better odds than one in 20). We expect the even smaller patch widths to achieve a high yield as well, eventually.

Figure 4.20 shows yield plots for the solid $3 \times 3 \times 3$ cube from Figure 4.10. The fully addressable solution here requires 27 species, the intermediate uses six, and the minimal solution only four. The trends seen in Figure 4.19 can be seen here as well, with the exception that the wider, more flexible patches now perform much better. Here the aggregation is mitigated by the cubes being more interconnected. While, without torsion, a single bond could rotate freely around its axis, the particles are held in place by multiple bonds along orthogonal axes, binding more strongly the more bonds they form.

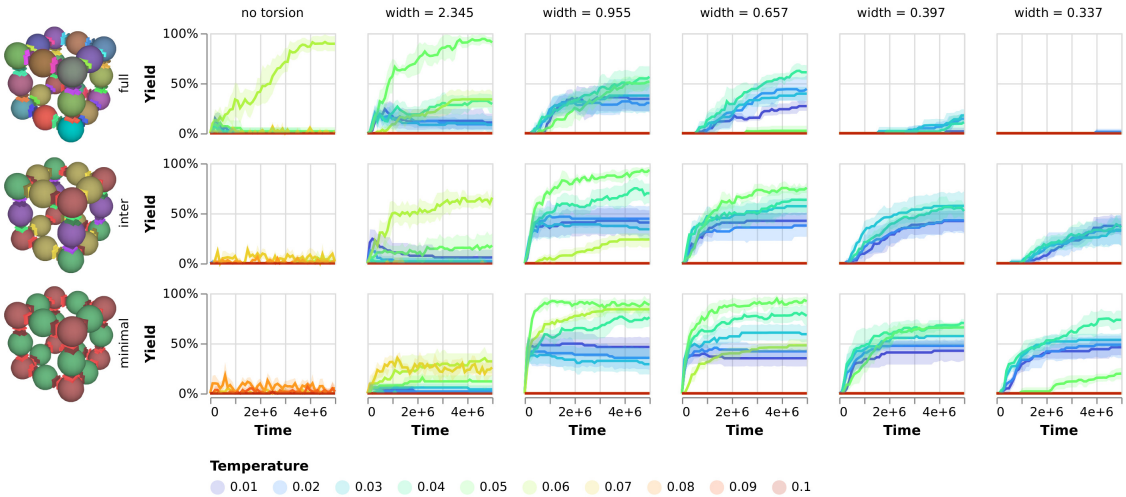


Figure 4.19: Assembly yield for hollow $3 \times 3 \times 3$ cube designs. The top row shows the fully addressable solution, using 20 species and 24 colours. The middle row shows an intermediate solution, using 4 species and 4 colours. The bottom row shows the minimal solution, using 2 species and 1 colour. Columns correspond to different interaction potentials, with the leftmost column showing wide patches without torsion. The remaining columns show torsional patches with decreasing patch width. Solid lines are mean values from 5 duplicate simulations, and error bands show the 95% confidence interval band. Each simulation has a 0.1 particle density.

Finally, Figure 4.21 shows the yield plots for the solid $4 \times 4 \times 4$ cube from Figure 4.11, for the fully addressable solution with 64 species and the minimal solution with only six species.

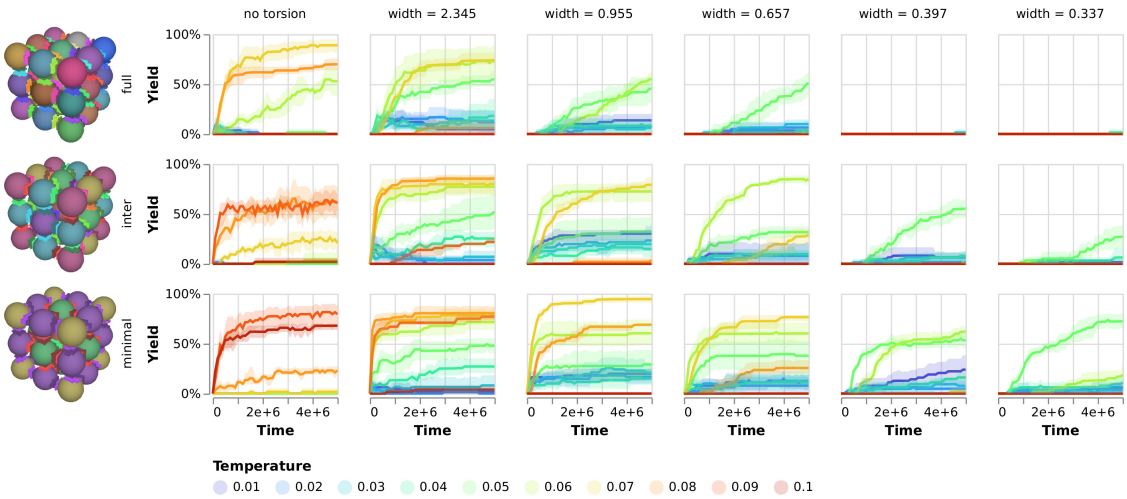


Figure 4.20: Assembly yield for solid $3 \times 3 \times 3$ cube designs. The top row shows the fully addressable solution, using 27 species and 54 colours. The middle row shows an intermediate solution, using 6 species and 9 colours. The bottom row shows the minimal solution, using 4 species and 3 colours. Columns correspond to different interaction potentials, with the leftmost column showing wide patches without torsion. The remaining columns show torsional patches with decreasing patch width. Solid lines are mean values from 5 duplicate simulations, and error bands show the 95% confidence interval band. Each simulation has a 0.1 particle density.

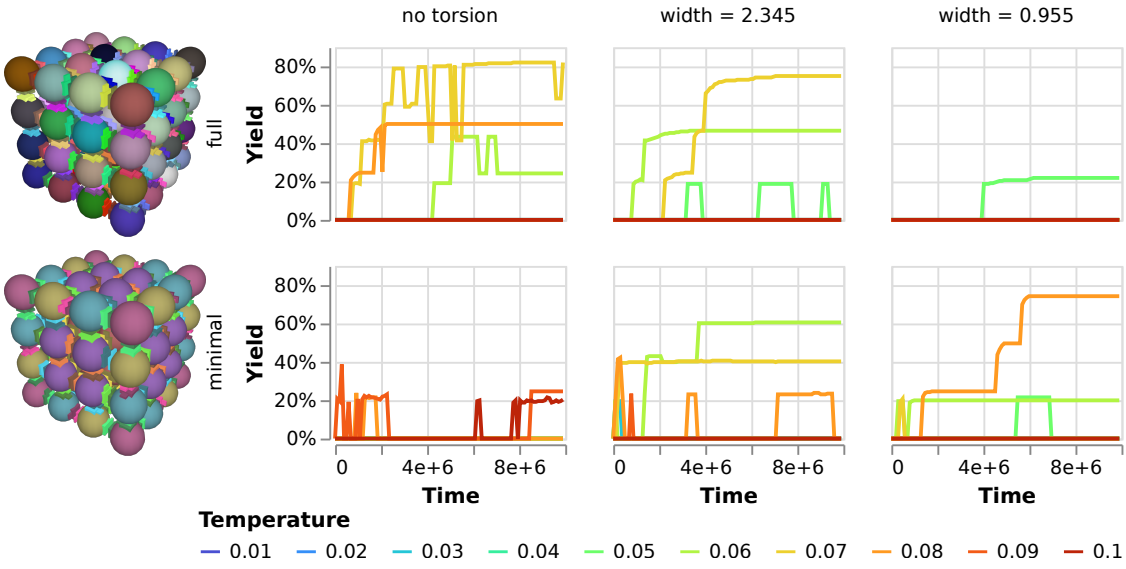


Figure 4.21: Assembly yield for two solid $4 \times 4 \times 4$ cube designs. The top row shows the fully addressable solution, using 64 species and 144 colours. The bottom row shows the minimal solution, using 6 species and 8 colours. Columns correspond to different interaction potentials, with the leftmost column showing wide patches without torsion. The remaining columns show torsional patches with decreasing patch width. Each simulation has a 0.1 particle density.

4.6 Multifarious assemblies

Another feature of the SAT solver approach is the ability to design *multifarious* assemblies, that is, rules that can assemble into more than one shape. This is done by defining multiple distinct shapes next to each other as input to the solver. As an example of this, we solve a "wolf" and a "human" shape first separately (Figure 4.22.a–b), then as a single "werewolf" shape specification (Figure 4.22.c).

The wolf shape consists of 14 cubes, while the human has 13, so they are roughly the same size. While the fully addressable multifarious solution ("werewolf") would require 27 species, the SAT solver approach found a minimal solution using only ten species and eight colours, with two species shared between both shapes.

We assemble the three minimal solutions (human, werewolf, and wolf) and compare the yields at which they assemble either human or wolf shapes. As seen in Figure 4.23 the individual human and wolf shapes assemble with 100% yield (at sufficiently low temperatures), while the werewolf design assembles with approximately 50% yield for either shape, with the wolf being favoured slightly at lower temperatures. The relative concentration of species in the werewolf simulations was chosen so that there was sufficient material to assemble either ten humans or ten wolves. Any chimeric clusters, with part human and part wolf, are not possible as such solutions have been ruled out by the stochastic assembler.

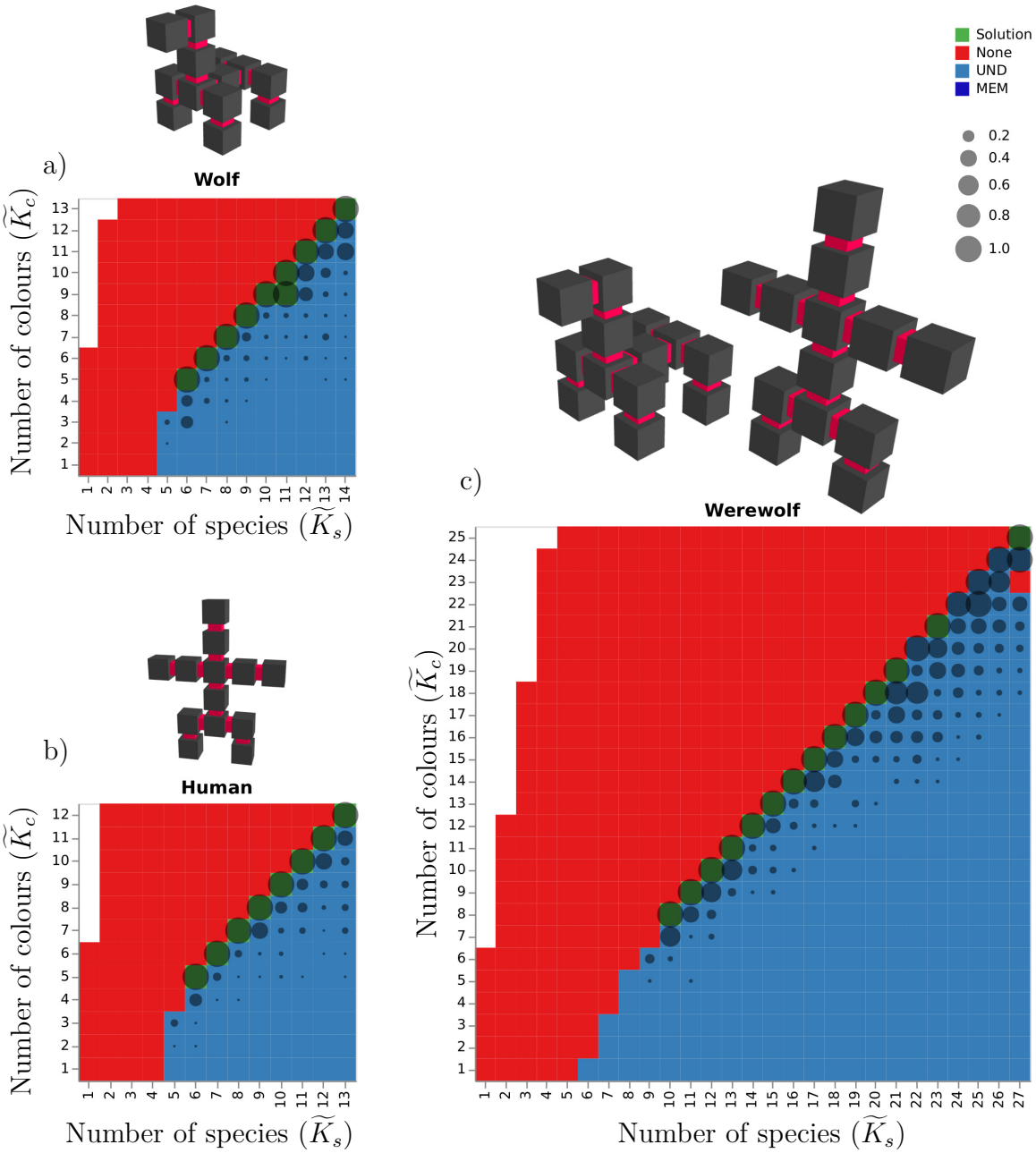


Figure 4.22: Designing a multifarious "werewolf" shape. **a)** Solution landscape and depiction of the individual "wolf" shape **b)** Solution landscape and depiction of the individual "human" shape. **c)** Solution landscape and depiction of the combined "werewolf" shape. The coordinates of both shapes are added as input to the SAT solver, thus producing solutions that can assemble both shapes.

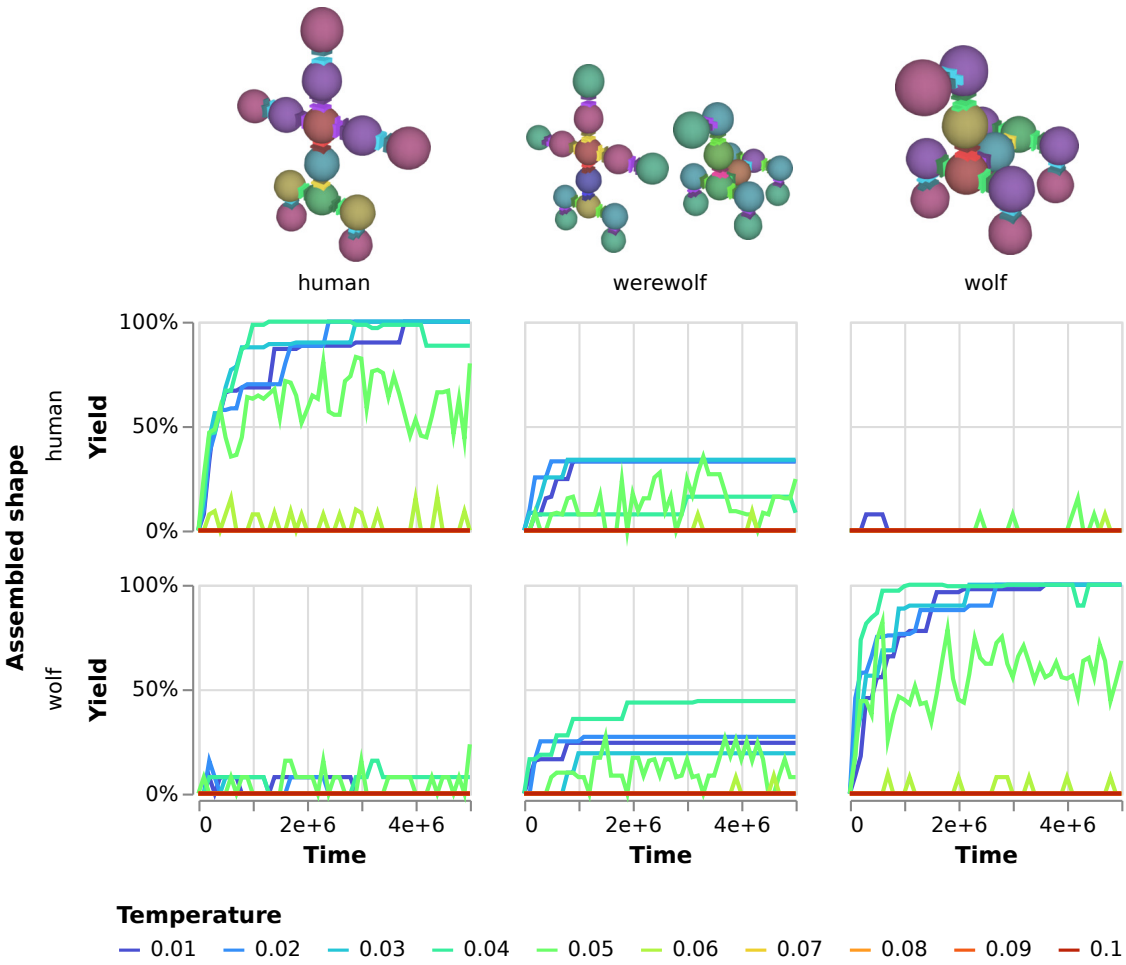


Figure 4.23: Multifarious werewolf assembly. The leftmost column shows the minimum solution for the "human" shape, the rightmost shows the minimum solution for the "wolf" shape, and the middle column shows the minimum solution that can reliably assemble both human and wolf shapes, here labelled "werewolf". The top and bottom rows show the yield at which the designs assemble the "human" and the "wolf" shapes respectively. Each simulation is done using the wide torsional patch potential with width = 2.346) at a 0.1 particle density.

4.7 Conclusion

While Chapter 3 showed how to map input rules into output shapes, this chapter has shown how to find the input rules that assemble a given shape, making it possible to determine the minimal complexity rule for an intended design (including multifarious designs). As was seen in Chapter 2, many current experimental works either use unique nanoparticles for each position in a final assembly or reuse particles in periodic or otherwise simple structures. The systematic design pipeline presented here should facilitate the design of bounded structures, saving resources

and minimising cross-talk between interfaces.

The design pipeline shown here can explore the landscape of assembly solutions for arbitrary shapes. In many cases, the minimum solution comprises significantly fewer particles than the target structure and assembles more rapidly than the solution in which every component is unique, with comparable or higher yields. Interestingly, the fact that, for highly specific binding domains such as those relevant for protein assembly, the minimal solution assembles better than more complex designs suggests an interesting evolutionary hypothesis: The recently discovered non-adaptive bias towards minimal protein complex designs [32] may produce complexes that will assemble better, providing an additional adaptive evolutionary driver towards simpler and therefore more symmetric structures.

For narrow (less flexible) patch interaction, the minimal solution assembles faster than the fully addressable solution. Hence, not only does the minimum assembly solution presents an advantage in terms of manufacturing costs for certain experimental realisations and conditions, but it is also predicted to better assemble than the fully addressable option. Generally, the kinetics of assembly is slower for narrower patch interactions, as random collisions are less likely to lead to successful bond formation. Additionally, for the fully addressable system, two randomly chosen particles are less likely to have compatible interfaces, and hence the structure growth is expected to be slower than for the minimal solution.

These results are likely to benefit the DNA nanotechnology field: the requirement for fewer unique components would mean significant savings in both time and resources, and we have developed a new automated design tool that can convert target structure design to a nucleotide-level coarse-grained DNA nanostructure representation for computational verification or for guiding an experimental design. The inverse design method is also applicable to other designed self-assembling systems, such as multi-component protein structures or coated nanoparticles [**zhu2021protein**, **xiong2020three**]. We note that for experiments, it may be desirable to incorporate specific building blocks, perhaps uniquely functionalised by

a specific material coating or attachment of a guest molecule, at specific locations. Our design method can be extended to allow for the inclusion of such constraints.

5

An introduction to design and simulation tools

Contents

While previous chapters have covered modular self–assembly on a very abstract level, approximating the modules as simple cubes or patchy particles, this chapter will introduce tools and methods for designing and simulating individual structures or modules folded using DNA (or RNA).

The following sections will cover a selection of practical design and simulation tools that have been developed over the years, providing context for the presentation of my contributions to the *oxView* tool in Chapter 6.

5.1 Design tools for DNA origami

Designing a DNA origami structure by hand would be very laborious for anything but the most simple design. A host of computer–aided design tools have been introduced over the years to make things easier. This section will cover some of the more common examples.

5.1.1 Lattice–based design tools

The caDNAno design tool [42] and the web–based scadnano [43] it inspired, allow the user to design DNA origami structures on a lattice of parallel helices.

caDNAno

CaDNAno [42] was introduced in 2009 as a way to simplify 2D and 3D DNA origami design. It has a graphical user interface with multiple panels, as seen in Figure 5.1. In the slice panel, the designer can place virtual helices on a lattice (either hexagonal or square), seen in the leftmost panel of the figure. The helices can then be filled in with strands and connected using crossovers in a path panel, seen in the middle of the figure.

Finally, caDNAno is also available as a plugin to the Autodesk Maya software [44], which enables a 3D visualisation of the design, as seen in the render panel to the right in Figure 5.1. However, caDNAno does not support Maya versions after 2015 [45].

Scadnano

Scadnano [43] (short for *scriptable caDNAno*) is a relatively new design tool, independent from but inspired by caDNAno version 2. The main difference is that Scadnano is entirely web–based (thus not requiring any installation). The python code base is also designed to make it easier to write scripts generating DNA designs. Scadnano can be found at `https://scadnano.org/`.

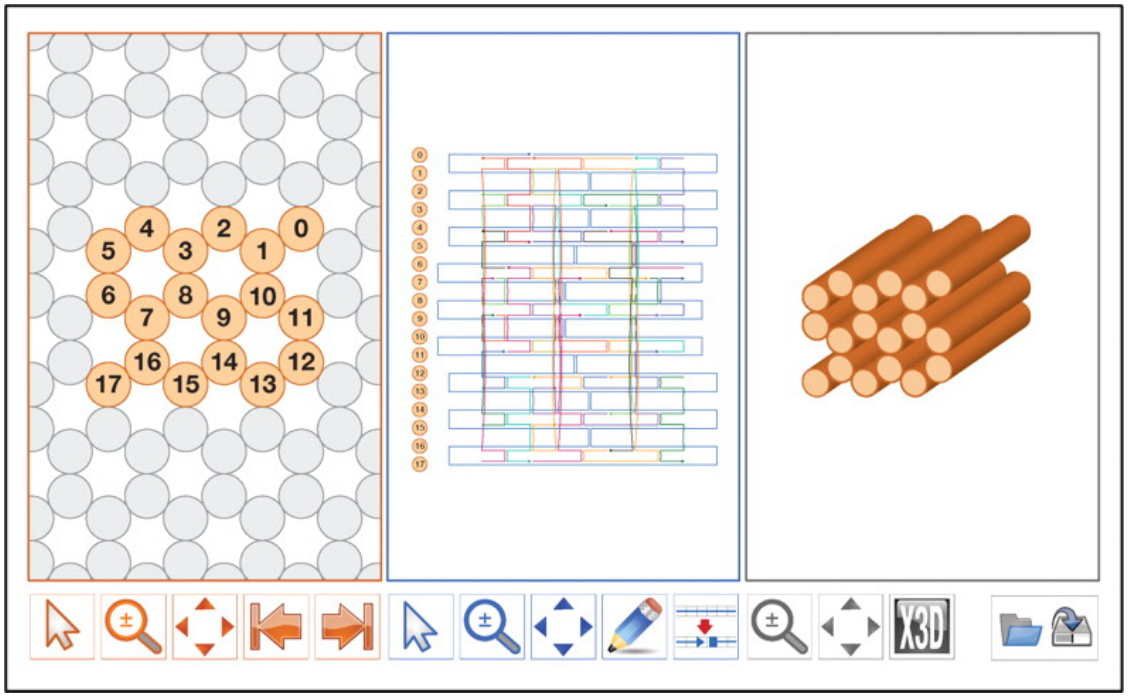


Figure 5.1: The caDNAno design interface, adapted from [42]. The slice panel (left) shows helices as circles on a lattice, while the path panel (centre) shows individual strands from a flatened side view of the helices. The rightmost panel shows a 3D visualisation of the design.

5.1.2 Top–down shape converters

While tools like caDNAno simplify bottom–up design, where the user builds structures from individual strands and nucleotides, a top–down tool can take a polyhedral target shape as input and provide a suitable origami design as output.

BSCOR

In 2015, Benson et al. published a method for converting arbitrary mesh designs into a DNA origami mesh [46]. Figure 5.2 shows a set of example polyhedral shapes, with the designed shape in Figure 5.2.a), the output DNA design in Figure 5.2.b), and microscopy characterisations in Figure 5.2.c-d). A follow–up paper in 2016 also introduced the ability to design flat–sheet meshes [47]. BSCOR uses single DNA duplex edges, with double edges added whenever topologically necessary.

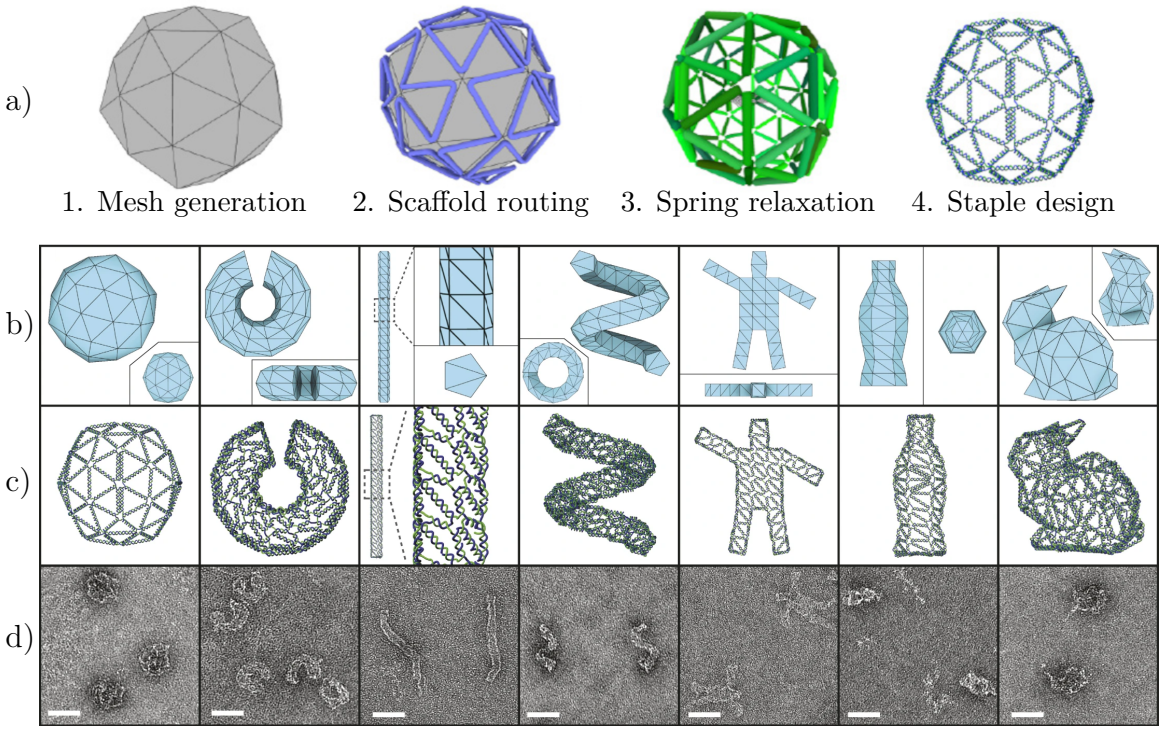


Figure 5.2: 3D meshes rendered in DNA origami using BSCOR. Adapted from [46] and [48]. **a)** Automated design process, where a scaffold is routed onto a mesh, each helix is relaxed using spring forces, and staple strands are added. **b)** Examples of initial meshes. **b)** Completed DNA designs with strands rendered as tubes. **c)** Negative–stain dry–state TEM micrographs of each design.

ATHENA

ATHENA is a recently published tool bundle for automatic designing wireframe origami shapes [49]. As seen in Figure 5.3, ATHENA brings together earlier software such as PERDIX, METIS, DAEDALUS and TALOS, into a single package capable of facilitating the design of both 2D and 3D wireframes with different edge designs.

5.1.3 Free–form or hybrid tools

The final category of design tools is either free–form, where designs are drawn without a lattice, or hybrid tools combining both lattice and free–form design.

Tiamat

Tiamat is an early free–form design tool (introduced in 2009) [50] running exclusively on Microsoft Windows. The second version can handle both DNA and RNA designs,

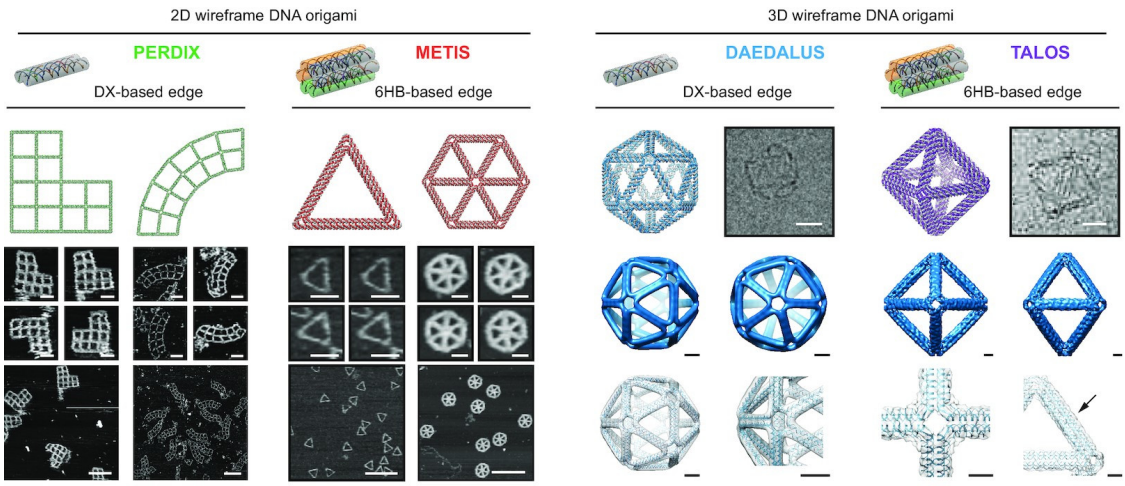


Figure 5.3: Automatic wireframe origami shapes using ATHENA. Adapted from [49]. ATHENA includes the previous design packages PERDIX, METIS DAEDALUS and TALOS for 2D and 3D wireframe design, using double crossover (DX) and six–helix bundle (6HB) edges, respectively.

either drawing them from scratch or importing them from PDB files. It can also export JSON format for easier conversion to other formats. See Figure 5.4 for a screenshot of the user interface of Tiamat 2, where a DNA tetrahedron is loaded.

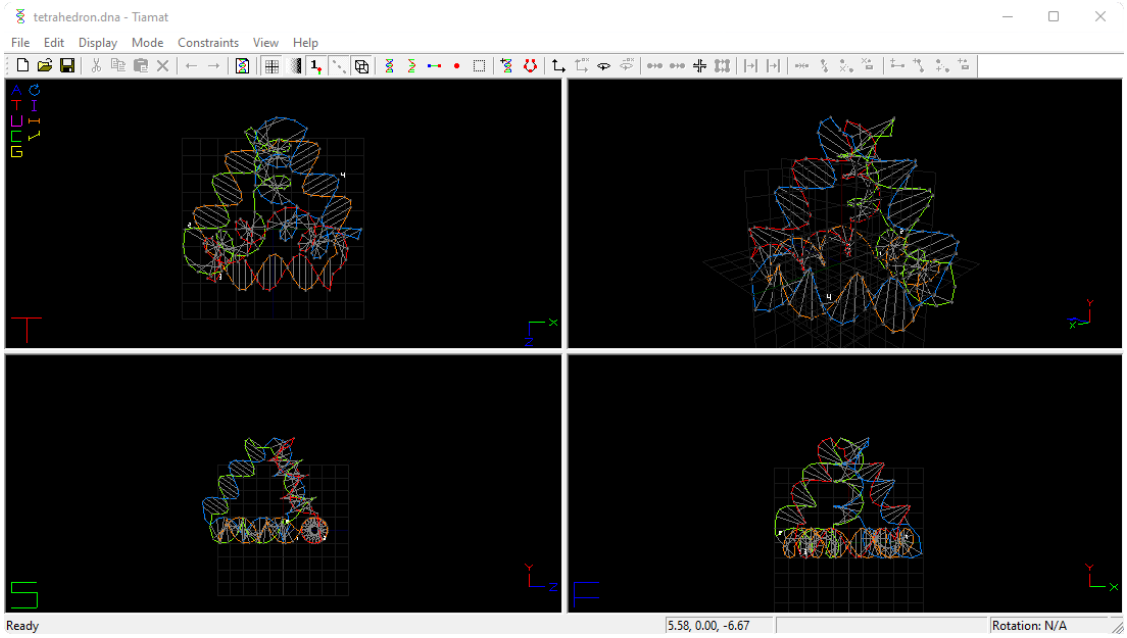


Figure 5.4: A screenshot of the Tiamat [50] (v2) user interface. The loaded tetrahedron design is from the Yan Lab resources page [51]

vHelix

The free–form tool vHelix [46] is a plugin for the commercial Autodesk Maya software [44] and was developed together with the BSCOR toolkit. Users can import the "rpoly" wireframe result from BSCOR, create designs from scratch, or import caDNAno files. It is also possible to export oxDNA simulation files. An example of the vHelix interface is shown in Figure 5.5.

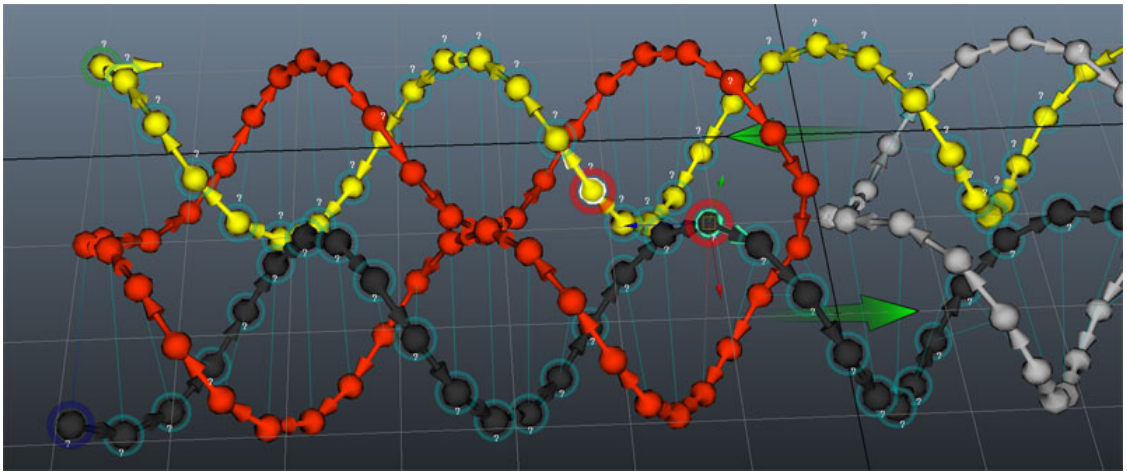


Figure 5.5: Free–form editing in vHelix. Image is from the vHelix website `http://www.vhelix.net/` [48].

Adenita

Adenita [52] is a hybrid free–form editing tool developed as a plugin to the commercial SAMSON connect toolkit [53]. As seen in Figure 5.6.a), the design can be visualised and edited at multiple levels of abstraction, from an all–atom representation, through nucleotides and strands to cylinders representing entire helices. Strands can be drawn on a lattice in a 2D view or edited in 3D. Figure 5.6.b) shows some of the available editing and visualisation tools. Note, for example, the wireframe creation tool based on the Daedalus algorithm. Adenita can also load caDNAno files and export oxDNA simulation files.

Like vHelix (Section 5.1.3), Adenita is tied to the editor for which it is a plugin, requiring multiple installations.

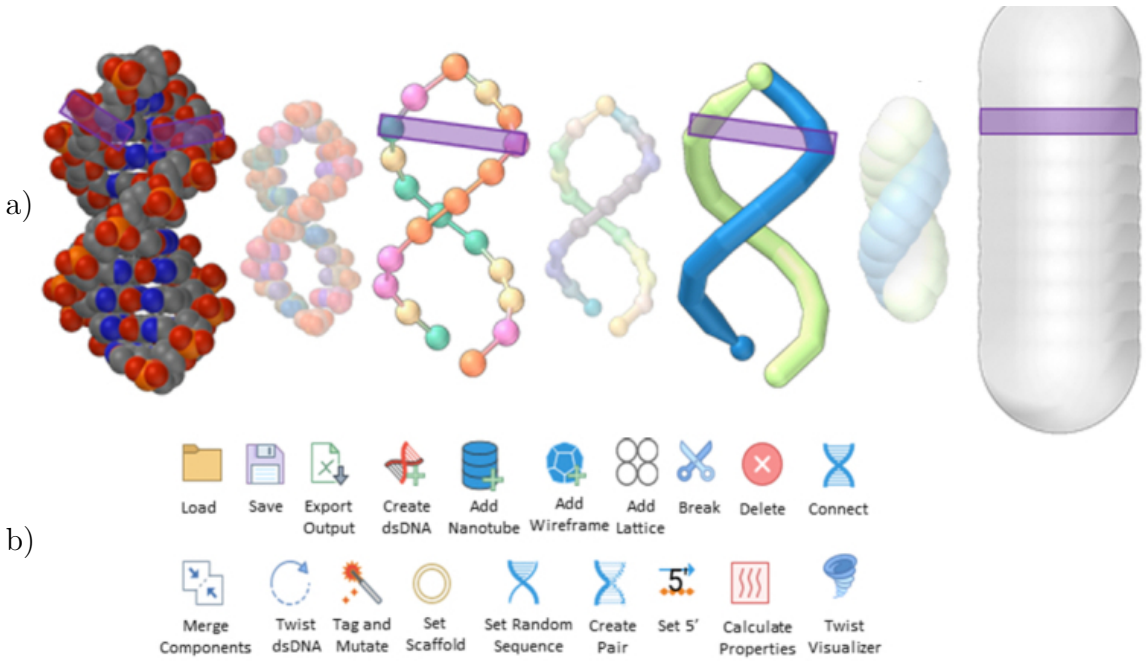


Figure 5.6: Adenita, adapted from [52]. **a)** A DNA double–helix visualised at different abstraction levels, with a purple band highlihghting a specific base–pair at all the four main levels. **b)** Available editing and visualisation tools in Adenita.

MagicDNA

MagicDNA [54] is another hybrid design tool using orientable lattices. As seen in Figure 5.7, it has a computer–aided design workflow where geometry can be specified from helix cross–sections or imported from a part library, then assembled and oriented into an integrated structure (using multiple scaffolds if necessary). MagicDNA is built as an application for the MATLAB software, so (like the previous two tools) installation requires a commersial third–party tool.

OxView

The oxView application [55, 56] was developed as part of this thesis project and will be described more in Chapter 6. Compared to the other design tools introduced here, oxView excels in combining designs from various formats, as well as easily preparing structures for oxDNA simulation (Section 5.2.2). Finally, oxView does not require any installation and is immediately available as a web–app at `www.oxview.org`.

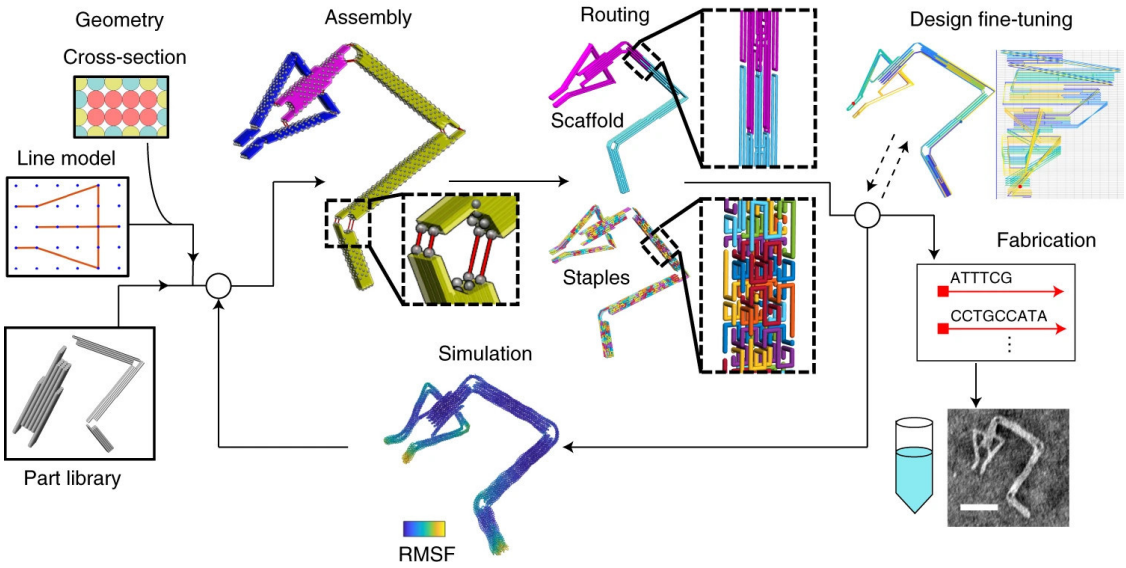


Figure 5.7: MagicDNA design workflow, adapted from [54]. Assemblies can be created from line drawings and helix cross sections or imported from a library of parts. After strand routing and some optional fine–tuning in caDNAno, designs can be exported as sequences for fabrication or oxDNA files for simulation.

5.2 Nucleic acid simulation models

Simulating a structure can both guide decisions at the design stage and provide insight for understanding experimental results. Coarse–grained models tend to run faster but may lose some accuracy compared to models with more detail. This section covers simulation models at an increasing level of coarse–graining, from individual atoms to cylindrical helices.

5.2.1 All–atom simulation

Simulation tools such as NAMD [57], use force fields such as AMBER [58] and CHARMM [59] to model interactions between individual atoms. While it is possible to perform atomistic simulations of large DNA origami structures [60] as shown in Figure 5.8, the simulations take a long time to run, and it is unknown how well the models represent DNA thermodynamics [61].

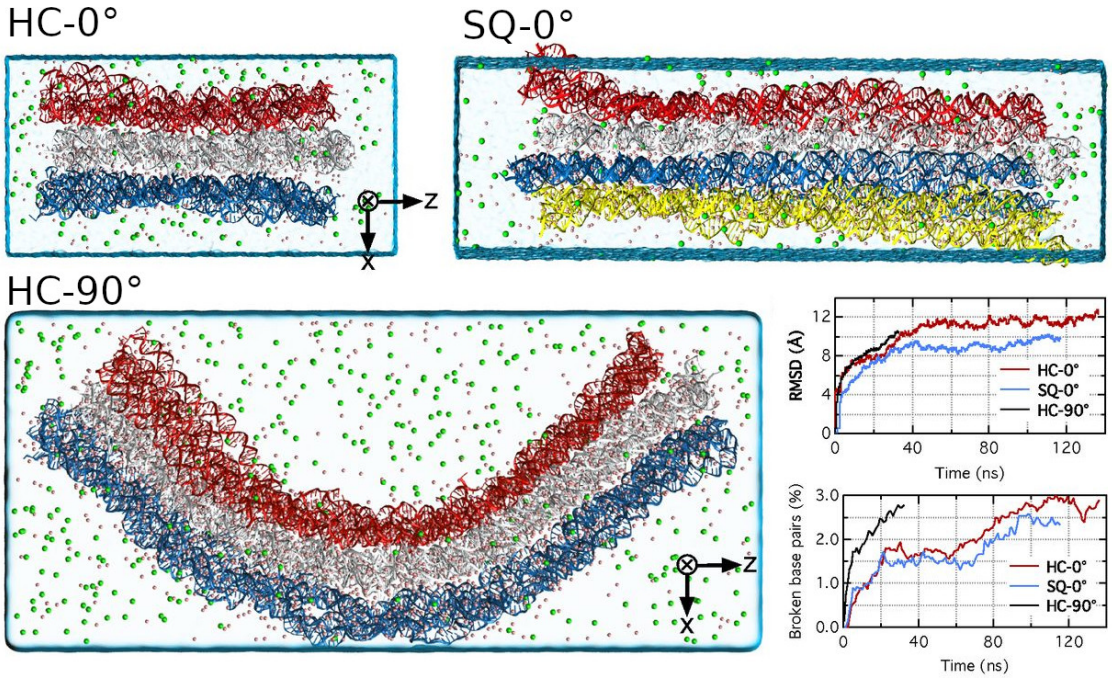


Figure 5.8: All–atom simulation of three DNA origami structures, adapted from [60]. HC-0° is a honeycomb (hexagonal) lattice origami with no programmed curvature, while SQ-0° is designed on a square lattice. HC-0° has a programmed 90° bend. The plots show the root–mean–square deviation from the original atom positions (top) and the fraction of base pairs broken during the simulation (bottom).

5.2.2 oxDNA/RNA

In 2010, a coarse–grained simulation software called oxDNA was introduced by Thomas Ouldridge and collaborators [62]. Rather than using a coarse–grained "bottom–up" definition, where the model parameters are inferred from the behaviour of a more detailed model, the oxDNA model is defined "top–down", trying to reproduce experimentally relevant properties [61]. The oxDNA model was calibrated to fit the Nearest Neighbour model [63], reproducing experimental duplex melting data, as well as to known structural parameters such as the persistence length. This has enabled the model to simulate complex DNA origami devices with a generally good agreement with experimental data [64]. Figure 5.9 shows the forces acting on the rigid-body nucleotide making up the oxDNA model.

oxDNA runs both Molecular Dynamics (MD) and Monte Carlo (MC) simulations of DNA modelled at the level of individual nucleotides. The Molecular Dynamics

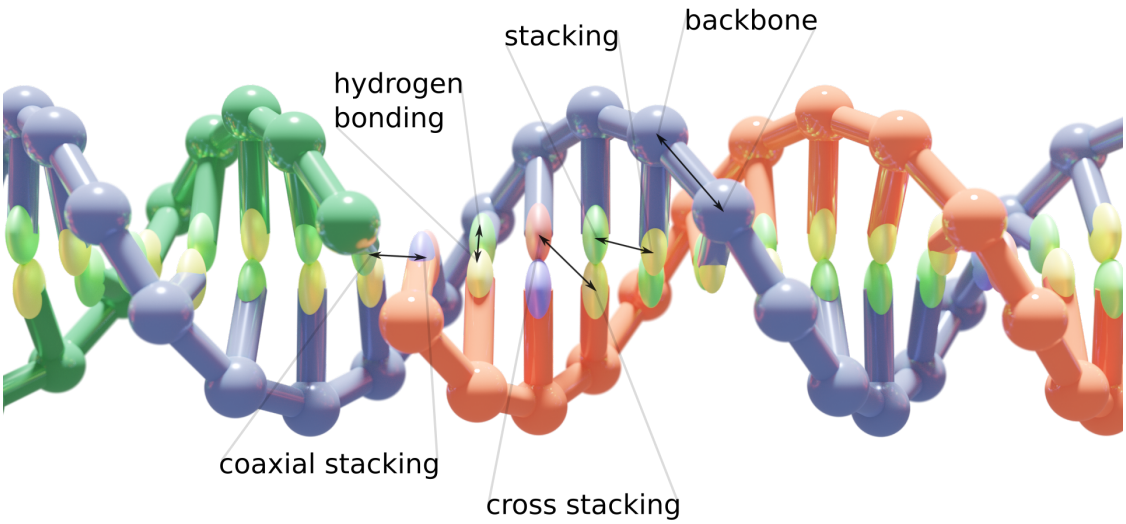


Figure 5.9: The interaction forces of the oxDNA model. Each nucleotide is modelled as a rigid body with four interaction sites; a backbone repulsion site, a base repulsion site, a stacking site, and a hydrogen bonding site. Apart from the five interactions annotated in the image, nucleotides also have an excluded volume created through the two repulsion sites. The image was created in oxView and rendered in Blender

simulations update the positions and orientations of the nucleotides according to Newton's laws of motion. Since there is no explicit solvent (that is, no water particles), the thermostat is used to ensure diffusive motion [61]. If two particles are very close together, the simulation is likely to become numerically unstable. This instability can be mitigated by using a smaller integration time step and a temporary (non–physical) limit on the maximum interaction force.

An alternative solution is to use the Monte Carlo dynamics, which instead updates the particles according to random moves sampled from the same Boltzmann distribution [61]. Particle overlaps can be removed by first simulating a system using MC dynamics (as the probability of overlaps are zero), thus "relaxing" the system in preparation for MD simulation [65]. Further relaxation can also be done in MD simulation using a maximum backbone force, gently shortening unnaturally extended backbone bonds.

Since its creation, both the model and the software running it has been extended

and improved [40, 66–69]. In 2014, the model was extended to include RNA by Petr Šulc and collaborators [70], showing its ability to model a set of common RNA motifs. In 2021, the model was further extended to include protein–DNA/RNA hybrids by Jonah Procyk and collaborators [71].

While oxDNA can be very useful for modelling a structure, it has traditionally not been very accessible for experimentalists. Although efforts have been made to create helpful tutorials [65], users still need an understanding of the command line to compile and run the code. This limitation changed in 2020 with the launch of the `www.oxdna.org` web server [72], where users can set up and analyse simulations through a simple web interface. Finally, the oxView interface [55, 56] (described in Chapter 6) has greatly facilitated the preparation, relaxation, and visualisation of oxDNA simulations.

5.2.3 mrDNA

The mrDNA simulation model [73] is a multi–resolution model representing a user–defined number of base pairs as a rigid–body bead.

Since mrDNA is implemented in Python and has a spline–based helix representation, users can write scripts to edit the structure, translating and rotating parts before starting the simulation. Thus, with some skill, topological issues or over–stretched bonds can be resolved even before starting to simulate.

A selection of the DNA designs I have relaxed, using both mrDNA and oxDNA, are shown in Figure 5.11. The main thing to note here is how the helices (shown as red lines) in the leftmost images are all parallel to each other (because of how they were designed in caDNAno). The mrDNA relaxation then allows the helices to quickly assume lower–energy positions and orientations, saving significant time compared to running oxDNA on the original configurations.

The first two examples, from [74] and [75] and illustrated in Figure 5.11.a) and 5.11.b), are both quite straightforward to relax in oxDNA, although the relaxation is much faster using mrDNA.

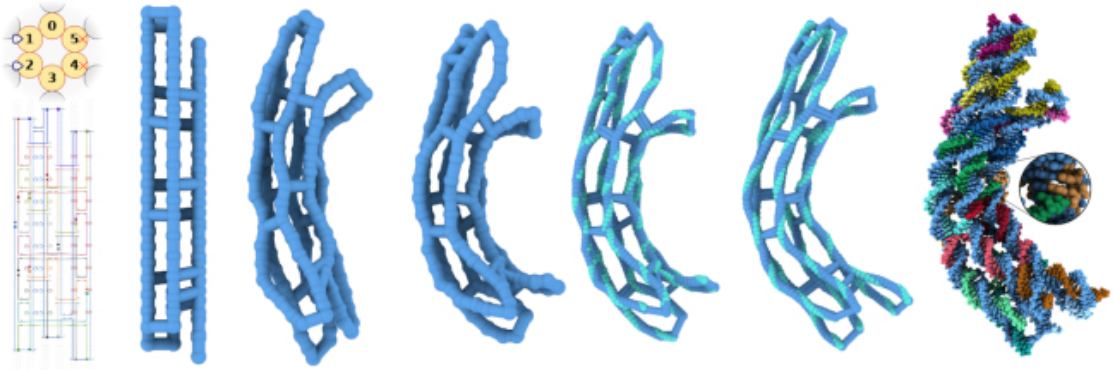


Figure 5.10: Illustration of a design being relaxed through multiple resolutions in the MrDNA model, adapted from [73]. From left to right, a caDNAno file is imported as a low-resolution bead model, then incrementally increased in resolution as more beads are added until one bead exists per base pair. The teal beads then represent the local orientation of the bases. Finally, the structure can be converted to an all-atom (rightmost image) or an oxDNA representation.

The tensegrity kite structure, from [76] is harder to relax since, as seen in the first image in Figure 5.11.c, the two helix bundles are drawn parallel to each other in caDNAno. Given enough time to relax, they should still become orthogonal, but a much more efficient way is to write a mrdna script to rotate one helix bundle so that it is orthogonal from the start, as seen in the middle image of 5.11.c. The remaining overstretched bonds are then quickly relaxed using mrdna.

Finally, the Möbius strip, from [77], is particularly tricky to relax, since the caDNAno design have all helices drawn in the same plane, with bonds from each end stretching through the whole structure and intersecting at a single point, as can be seen in the first image of Figure 5.11.d. With some help from Chris Maffeo, however, I was able to use a mrdna script to edit the structure into a configuration much easier to relax, as seen in the second image of Figure 5.11.d. Since the caDNAno design does not make it clear whether the Möbius strip should be left–handed or right–handed, this is also decided in the script; changing the rotational direction will produce a mirrored version of the structure, as seen in the third image of Figure 5.11.d.

Based on the usefulness of scripted transformations, I created a rudimentary interactive editor interface to mrdna to also allow for visual editing. However, it would need significant refinement to be externally usable. More importantly,

the oxView editor, described in Chapter 6, can now quickly resolve relaxation issues such as those exemplified above, both using clustered rigid-body dynamics (Section 6.2) and manually (Section 6.3).

5.2.4 Cando

Cando is a finite element modelling framework [78, 79] available through a web server at `https://cando-dna-origami.org`. DNA double helices are modelled as elastic rods (connected by rigid crossovers) that stretch, twist and bend in line with experimental measurements. See Figure 5.12 for a set of example structures designed in caDNAno and simulated in Cando.

Similar to mrDNA, Cando can be used to relax caDNAno designs faster than oxDNA (since it uses a less detailed model). However, the resulting output is not as convenient as an oxDNA input configuration. While an option exists to include an atomic model output in the PDB format (which can be converted to oxDNA simulation files), this requires a caDNAno sequence file and is only possible for designs with at most 10,000 base pairs.

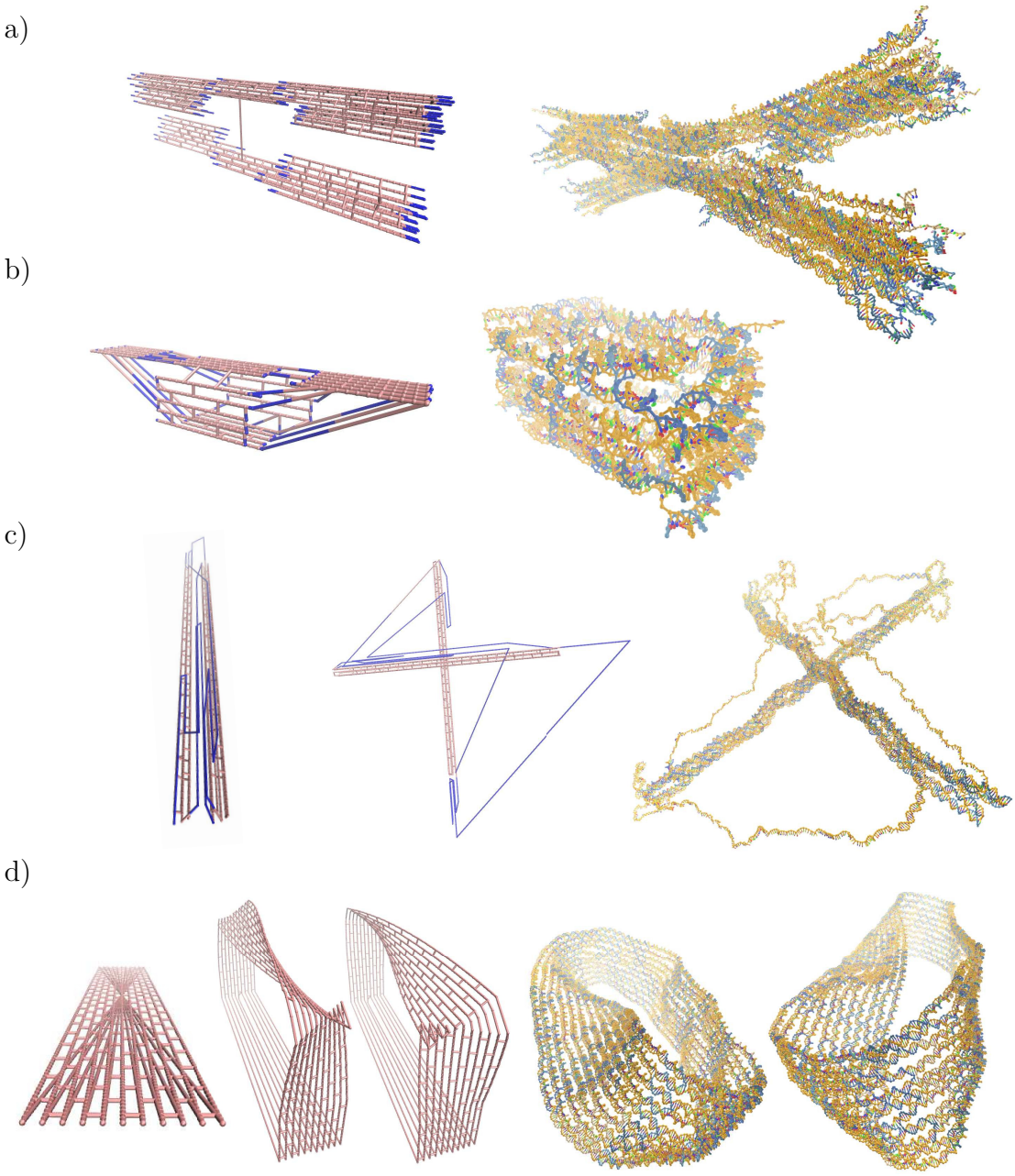


Figure 5.11: Relaxation results for various DNA designs. Each row depicts a new design, with the left–hand side showing the structure as it was drawn in caDNAno (and parsed by mrdna), while the right–hand side is the relaxed structure in oxDNA. Intermediate images are edits done in mrdna. While the switch design [74] in **a)** and the small DNA origami box [75] in **b)** relaxed without any required editing, the tensegrity kite structure [76] in **c)** and the Möbius strip [77] in **d)** benefited greatly from moving selected helices to a position off the lattice before starting the simulation.

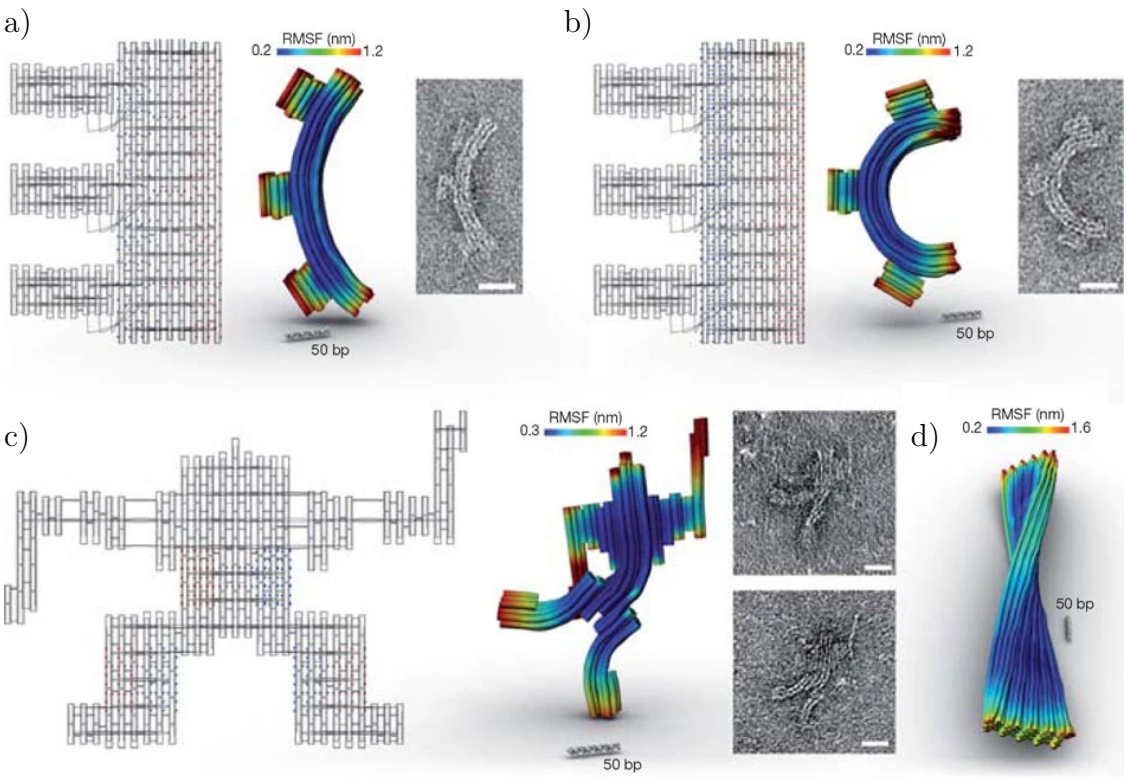


Figure 5.12: Cando simulation results. Adapted from [78]. caDNAno design diagrams, Cando structure and flexibility prediction, and negative–stain TEM micrographs **a)** 90 degree gear. **b)** 180 degree gear. **c)** "Robot" design **d)** 60-helix bundle.

6

Structure design and analysis in oxView

Contents

This chapter contains my results on my second project: how to design, simulate and analyse DNA and RNA nanostructures.

As I started this DPhil, I was tasked with the issue of converting origami designs created in caDNAno (see Section 5.1.1) so that they could be correctly simulated in oxDNA (see Section 5.2.2). You have seen some of the tools I tried introduced in Chapter 5. I collaborated with Hannah Fowler from the Doye group at Theoretical

Chemistry, who simulated an extensive collection of DNA designs. This was a good opportunity to investigate why some structures were more problematic to convert and relax than others.

At this time, the available method for converting a caDNAno design into the oxDNA format was to use an old python script included in the UTILS directory of the oxDNA repository. However, the script was not easy to use, and it failed for many structures. At the end of 2018, the tacoxDNA webserver [80] was launched, updating the conversion script and making it more accessible.

Still, since caDNAno structures are drawn on a lattice, where all helices have to be parallel to each other (as was shown in Figure 5.11), the resulting oxDNA configurations often had unnaturally extended backbone bonds, requiring time–consuming relaxation.

During a secondment within the Šulc group at Arizona State University in 2019, I was able to contribute to their development of a web–based oxDNA viewer called oxView [55, 56]. This tool can now be accessed at `www.oxview.org`.

Among the main early features that I added to oxView was a cluster–level rigid–body dynamics option (detailed in Section 6.2) that, in many cases, speeds up the relaxation by orders of magnitude compared to oxDNA relaxation alone. Since then, I have collaborated with the Šulc group to add more features and to make the tool more accessible as a visualiser and editor.

For our second oxView publication [56], I rewrote the main parts of the taxoxDNA codebase into TypeScript, resulting in the *taxoxdna.js* library (`https://github.com/Akodiat/tacoxdna.js`) which oxView now uses to import various standard design formats (including caDNAno) automatically. This is described in Section 6.1.

This chapter will present my own contributions to oxView [55, 56] unless otherwise stated. However, I wish to acknowledge the work done by Erik Poppleton and Michael Matthies; without them, this tool would not exist. Michael was the original oxView developer and has done great work in supporting live oxDNA relaxations through the *ox-serve* webserver. Erik enabled oxView to render and analyse systems with over a million nucleotides and has created a large set of useful

analysis scripts. Jonah Procyk has also made significant contributions to the oxView codebase by incorporating tools for working with proteins. Likewise, Aamik Mallya has helpfully contributed to helix creation and colouring.

6.1 Importing designs

There are many different formats available for DNA origami design; some of the main design tools producing them are covered in Section 5.1. This section will describe how to use oxView to import designs from such tools.

6.1.1 Basic import

Thanks to the *tacoxdna.js* library, importing designs is now generally straightforward. An import button in the "file" menu allows the user to import caDNAno, rpoly and tiamat (json) files, while oxDNA and PDB files can be directly opened. Some additional formats still require the use of the external Tacoxdna webserver [81]. Any loaded structure can then be exported for oxDNA simulation, as described in Section 6.5.

Importing caDNAno files

JSON files created using caDNAno (described in Section 5.1.1) can now be imported directly into oxView. See the import dialog shown in Figure 6.1.a). First, select the file to import and make sure to also select *caDNAno* as the file format. Next, choose the correct lattice–type; either *Square* or *Hexagonal*. Optionally, input a sequence to assign to the origami scaffold (which will otherwise be random).

Another option is to use the tacoxdna webservice to convert the caDNAno design into oxDNA files and then load those into oxView. However, designs loaded directly into oxView have the benefit of including correct colouring, clustering and base–pairing information, which would otherwise have been lost.

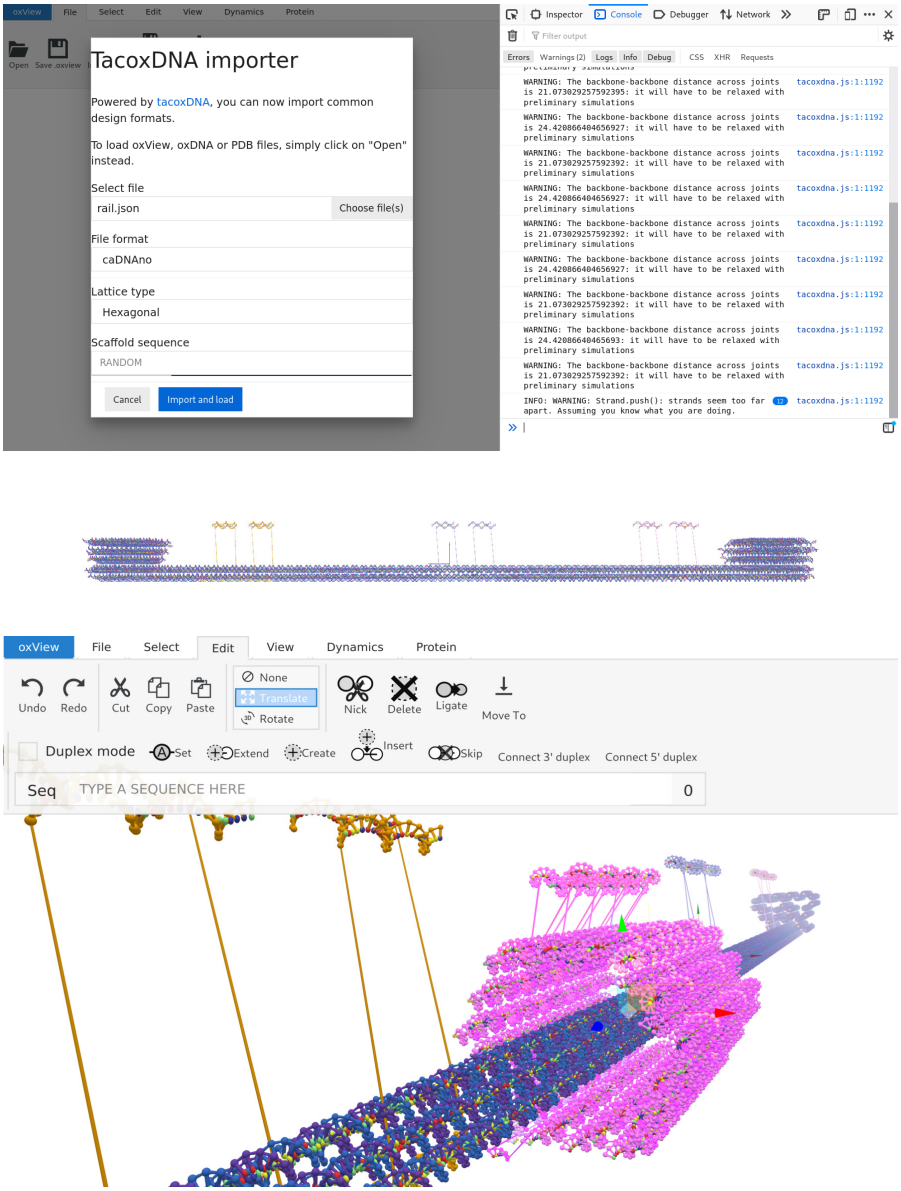


Figure 6.1: Importing caDNAno structues into oxView. **a)** The tacoxdna.js library import dialog in oxView (left), seen here importing a caDNAno design. Note the web browser console shown to the right, where any additional output from the import is written. **b)** Imported linear actuator rail design from [82]. **c)** Complete linear actuator structure from [82] assembled in oxView, after also importing the slider caDNAno design.

Importing rpoly files

Rpoly files are the output from the BSCOR [46] tool (described in Section 5.1.2) for converting polyhedral meshes into DNA origami. Select the rpoly file to import, making sure that *rpoly* is selected as the file format. Optionally, input a sequence

to assign to the origami scaffold (which will otherwise be random).

Importing Tiamat files

Tiamat (described in Section 5.1.3) designs in the *.dnajson* format can also be imported. Select file to import, making sure that *tiamat* is selected as file format. Binary *.dna* Tiamat files need to be reopened in Tiamat and saved to the text–based *.dnajson*. Select Tiamat version (1 or 2), then select the nucleic acid type (DNA or RNA).

By default, nucleotides without assigned base types will be given a random type. However, it is also possible to select a fixed default base.

Importing PDB files

While I did include the DNA PDB to oxDNA converter from Tacoxdna in *tacoxdna.js*, a more versatile PDB import was created by Jonah Procyk to support his ANM–oxDNA model [71]. Simply drag and drop (or load) a PDB file into an oxView window and the DNA, RNA and/or protein it contains will be automatically converted and loaded.

6.1.2 Multi–component designs

Designs spread across multiple files (or even multiple design tools) can be easily combined in oxView by simply importing them all and using the editing tools to arrange and connect them properly. Figure 6.1 shows an example of this, where the a slider design (Figure 6.1.c) is added and positioned relative to the already imported rail (Figure 6.1.b), both structures from [82].

6.1.3 Far–from–physical caDNAno designs

As mentioned in Section 5.2.3 and at the beginning of this chapter, some structures drawn in caDNAno will have a very far–from–physical configuration. Since it is only possible to draw all helices parallel to each other (on a lattice) in caDNAno, backbone bonds may be very elongated, creating high energies and/or topological problems.

The oxDNA software already includes relaxation procedures for such structures [65], bringing them together in a slow and controlled manner using a specified maximum backbone force. However, for large structures, this can take a very long time, even while using GPU simulation.

As discussed in Section 5.2.3 and shown in Figure 5.10, the mrDNA simulation model [73] can help in such cases. However, the easiest option now tends to be using oxViews editing and rigid–body dynamics capabilities. The following sections will explain how.

6.2 Rigid–body dynamics

For rapid relaxation of imported caDNAno designs, oxView includes so–called rigid–body dynamics. When the dynamics are active, clusters of nucleotides are treated as rigid bodies in a simple physics simulation, using the method described in [83] and running locally in the web browser.

Clusters can be defined manually, or by running an automatic DBSCAN [84] clustering algorithm. If caDNAno files are imported directly into oxView, they will already have clusters defined. Similarly, copying and pasting structures create separate clusters.

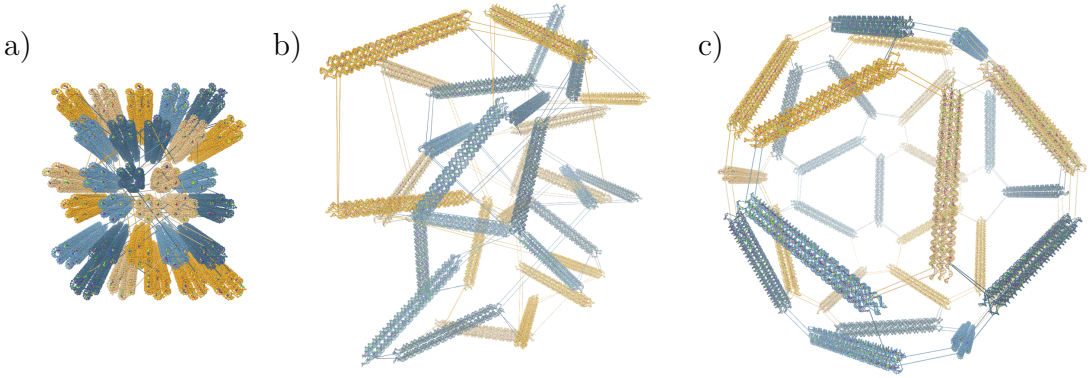


Figure 6.2: Rigid–body dynamics of clusters. Snapshots from the automatic rigid–body relaxation of an icosahedron, starting with the configuration converted from caDNAno **a)**, through the intermediate **b)** where the dynamics are applied, and **c)** the final resulting relaxed state.

Within the dynamics, clusters are held together with spring forces at each shared backbone bond, with a magnitude of $f_{\text{spr}} = c_{spr}(l - l_r)$ where c_{spr} is a spring constant, l is the current bond length and l_r is the relaxed bond length. To avoid overlaps, a simple linear repulsive force, of magnitude $f_{\text{rep}} = max\left(c_{rep}\left(1 - \frac{d}{r_a + r_b}\right), 0\right)$ is added between the centre of each group, where c_{rep} is a repulsion constant, d is the distance between the two centres of mass, and $r_a + r_b$ is the sum of the group radii (the greatest distance they can be while still overlapping).

As seen in Figure 6.2, the separate clusters will relax into a more natural shape, prepared for oxDNA simulation or further design. In some cases, however, simply moving the clusters manually (rigid–body manipulation) might be an even easier solution, which is covered in Section 6.3.

Another option for oxView dynamics is to use oxServe, a server developed mainly by Michael Matthies that connects oxView to an oxDNA instance using WebSockets. By first relaxing the clusters and then using oxServe to relax the structure on a nucleotide level, users can interactively perform the entire relaxation process from oxView.

6.3 Editing designs

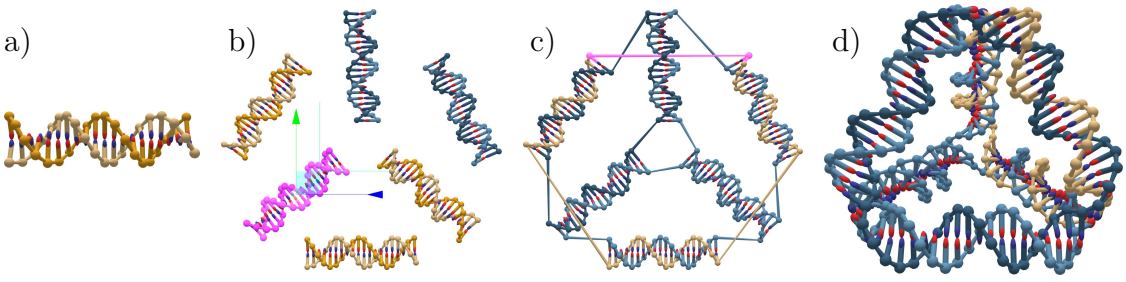


Figure 6.3: Designing the DNA tetrahedron from [85] using the oxView editing tools. **a)** An initial 20 base pair helix created. **b)** Duplicated helices being rotated and translated into place. **c)** Strands ligated together. **d)** The resulting 3D tetrahedron shape, as seen after applying rigid–body dynamics.

While oxView started as a visualisation tool, it has since been extended with a number of editing features. One of the earliest was the ability to perform rigid–body manipulation of selected nucleotides by dragging them with the mouse. I contributed

by adding transformation gizmos that simplify translation and rotation of selections using on–screen arrows and arcs for the user to manipulate. See Table 6.1 for a complete list of the editing tools currently available in oxView.

With the ability to create, remove, connect, and disconnect nucleotides, oxView users can now design structures from scratch. See, for example, Figure 6.3, where the DNA tetrahedron from [85] is created. The user first creates an initial helix (Figure 6.3.a) by typing a 20-base sequence and clicking the "Create" button in the "Edit" menu. Note that the "Duplex mode" need to be active in order for the complementary strand to be created automatically. Next, the user can copy and paste the helix repeatedly, using the "Translate" and "Rotate" tools to position the helices as seen in 6.3.b).

Any edit can also be undone and redone, either pressing Ctrl+Z or Ctrl+Z, or by pressing the corresponding buttons, or .

6.4 Visualization options

Depending on the design, a user of oxView might want to modify the visualisation settings to make certain features of the structure more or less visible.

Centring with periodic boundary conditions A useful utility when visualising an oxDNA trajectory is to keep the structure centred at the origin, stopping it from drifting out of view. I used the method described in [86] to achieve centring while taking periodic boundary conditions into account.

In essence, for each coordinate $p_j = \left(p_x^j, p_y^j, p_z^j\right)$ in the centring set of size n, each dimension $i \in \{x, y, z\}$ gets averaged in its own variable α_i, representing its 1D interval as a 2D circle (where the circumference b_i is the bounding box side length):

$$c_i = \frac{1}{n} \sum_{j=1}^{n} \left[\cos\left(\alpha_i^j\right), \sin\left(\alpha_i^j\right)\right]$$

Here, $\alpha_i^j = \frac{2\pi}{b_i} p_i^j$ is the angle on the unit circle and c_i is the average 2D position representing dimension i. Finally, the averages are converted back to cartesian coordinates:

Tool	Description
	**Create** a new strand from a given sequence. Select *duplex mode* to instead create a helix.
	**Copy** the selected elements (Ctrl+C).
	**Cut** the selected elements (Ctrl+X).
	**Paste** elements from clipboard (Ctrl+V to paste in original position, or Ctrl+Shift+V to paste in front of camera).
	**Delete** all currently selected elements (delete).
	**Ligate** two strands by selecting the 3' and 5' endpoint elements to connect (L).
	**Nick** a strand at the selected element (N)
	**Extend** strand from the selected element with the given sequence. Select *duplex mode* to also extend the complementary strand.
	**Insert** (add) elements within a strand after the selected element.
	**Skip** (remove) selected elements within a strand.
	**Rotate** selected elements around their center of mass (R).
	**Translate** currently selected elements (T).
	**Move to**. Move other selected elements to the position of the most recently selected element.
	**Connect 3' duplex**. Connects the 3' ends of two selected staple strands with a duplex, generated from the sequence input.
	**Connect 5' duplex**. Connects the 5' ends of two selected staple strands with a duplex, generated from the sequence input.
	**Set** the sequence of currently selected elements. Select duplex mode to also set the complementary sequence on paired elements.
	**Get**. Assigns the sequence of selected bases to the sequence input.
	**Reverse complement**. Generates the reverse complement of a provided sequence.
	**Search**. Highlights the position the provided sequence in each strand, if present.

Table 6.1: Editing tools available in oxView

$$cm_i = \pi + \frac{b_i}{2\pi} atan2(-c_{i,y}, -c_{i,x})$$

Change component sizes Depending on the structure scale, some visual components might be more or less common. This can be configured using the 'Visible

components' option in the "View" menu, where the nucleoside sphere, nucleoside connector, backbone sphere, and backbone connector components can be rescaled or hidden entirely.

Colours Nucleotides can be coloured by system, cluster, strand or by custom colouring specified by the user or at import. Users can also configure the default color palette used in oxView. Finally, it is also possible show colour overlays from the analysis scripts, mostly developed by Erik Poppleton.

Virtual reality Understanding the 3D structure of a design shown on a 2D screen is not alwasy so easy. Some view options like enabling fog might help the depth perception a bit, but oxView is also compatible with webVR, enabling the user to inspect their design with a virtual reality (VR) headset. The button to enable VR can be found in the "View" menu.

6.5 Exporting designs

Once a structure has been created in or loaded into oxView, it can be exported in a variety of formats.

6.5.1 Exporting oxDNA simulation files

OxView can export topology, configuration, and external force files for oxDNA simulation. Simply click the "Export oxDNA" button in the "File" menu and select the required file types.

One important thing to note is that the oxDNA format requires nucleotides to be correctly sorted with consecutive indices. Furthermore, this sorting is done, against convention, in a 3' to 5' order. Meanwhile, to facilitate editing, oxView nucleotides keep their indices even if the topology changes. Thus, nucleotides indices may be reassigned on oxDNA export, so it is important to export all files (topology, configuration, and forces) if the design has been edited.

6.5.2 Exporting other 3D formats

File formats such as glTF and STL contain geometrical information that can be 3D printed or imported into other 3D software such as Blender. OxView exports the scene as it is at the moment of export (at the current frame if a simulation trajectory is loaded), so it is important to configure intended component sizes and colours beforehand.

STL is an old and common standard for 3D shapes, containing only vertex coordinates (no colours). It is a popular input for 3D printing, but the file size is relatively large.

glTF (or glb if binary) is a modern standard for 3D scenes, storing geometry, hierarchy, and even material properties.

6.5.3 Exporting sequence files

Strand sequences can be exported as standard CSV files using the "Sequence file" export button in the "File" menu. The designed sequences can then be ordered and assembled experimentally.

6.5.4 Saving image files

It is possible to save an image of the current oxView view by simply clicking the "Save image" button in the "File" menu. This is a preferred option over a screenshot for two reasons. First, it is possible to increase the resolution by a scaling factor found in the "Image size" dropdown (rescale the browser window to change the aspect ratio of the image). Secondly, the background of the saved image will be transparent, simplifying further editing and composition.

For cover art and photo–realistic renders, it is also possible to export and load a glTF file into (for example) Blender, as seen in Figure 6.4. Note that in current Blender versions (2.9), large structures take a long time to import. So, make sure to disable any components not needed before export.

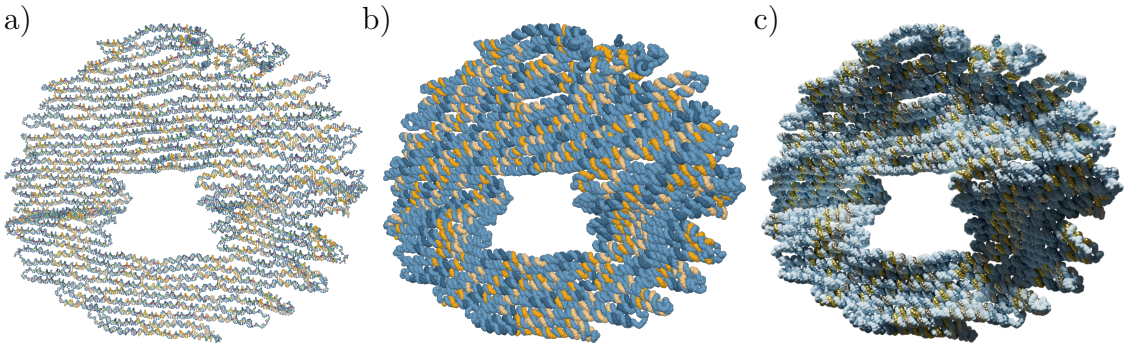


Figure 6.4: Component scaling and image export. **a)** Default oxView visualisation. **b)** Custom component scale and visibility. Using the"Visible components" dropdown in the "View" menu, the backbone spheres have been scaled up by a factor of 4.18 while all other nucleotide components have been hidden. **c)** The scaled scene in **b)** exported as glTF and rendered using Cycles in Blender.

6.5.5 Creating videos

One of the earlier features I implemented in oxView was the ability to create videos from oxDNA simulation trajectories. It is also possible to create lemniscate videos of single configurations, where the camera moves around the structure in a lemniscate–shaped loop. The video export uses the `CCapture.js` library, grabbing the `Three.js` canvas and outputting either "webm" or "gif" animations, or each frame as separate "jpg" or "png" images.

Click "Create video" in the "File" menu, choose video type (trajectory or lemniscate), file format, and frame rate. For lemniscate videos, it is also possible to set a video duration.

For trajectory videos, the camera can be manually moved while the video is being recorded, thus showing the simulation from different angles.

Another option for video creation is to follow the steps described in Section 6.5.4. When the structure is loaded into Blender, the camera can be animated (or the design rotated) using keyframes to render high–quality videos. To render an oxDNA simulation, use the "traj2blender.py" script, found at `https://github.com/Akodiat/traj2blender`, to automatically load keyframes corresponding to simulation steps.

6.6 Summary of oxView contributions

A summary of the main oxView features that I have personally implemented is provided here:

- Rigid-body dynamics relaxation of clustered origami components.
- Improved editing with an undo/redo stack, as well as transformation gizmos.
- A re-implementation of the tacoxDNA set of converter scripts allowing native import of structures into oxView.
- The option to export videos of simulation trajectories (and static configurations).
- A more user-friendly interface, based on a "ribbon" menu design from the Metro4 UI library.
- The option to export glTF files for use in other 3D software, including photo-realistic rendering tools.
- Virtual reality visualisation using webXR.

6.7 Converting RNA origami designs

During my secondment at the Andersen lab in Aarhus, I worked with converting RNA structures designed using their ASCII–based blueprint format into oxRNA simulation files. Examples of converted structures are shown in Figure 6.5. The Andersen lab has scripts for parsing their blueprint files and building the corresponding PDB structures (the first two columns of Figure 6.5). However, the resulting PDB files are not relaxed and would take a long time to relax using all–atom simulation. During the seconment, I modified the tacoxDNA [80] PDB parser to enable PDB–to–oxRNA conversion, but with the oxView PDB import now fully working, it is simply a matter of dropping the PDB files directly into oxView. The third column of Figure 6.5 shows the structures relaxed and simulated in oxRNA, some significantly different from the previously available PDB models in the second column.

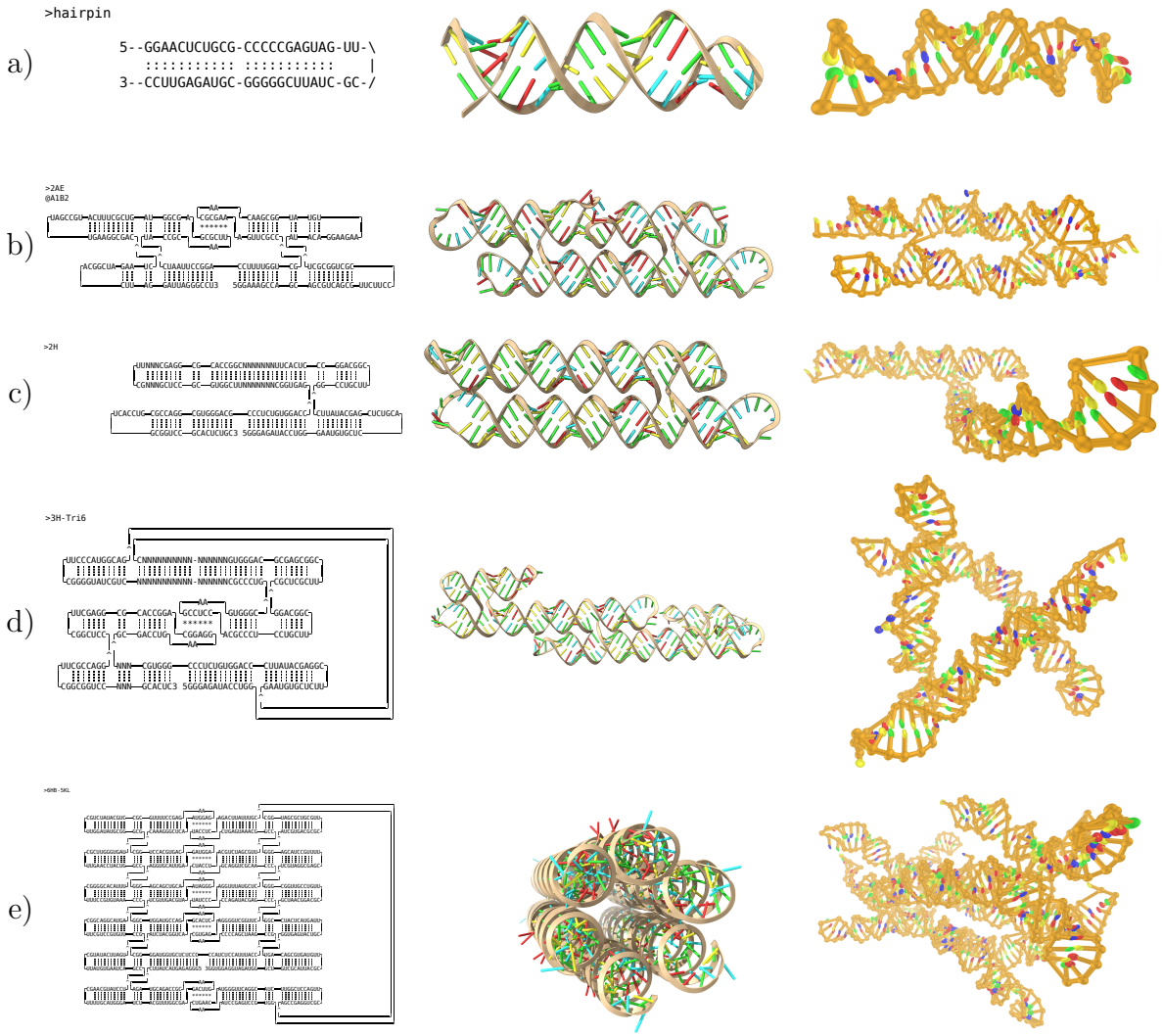


Figure 6.5: Conversion and simulation of various RNA designs. Each row, from left to right, shows the ASCII blueprint design, the PDB model (visualised using ChimeraX), and a frame from the simulated structure (visualised using oxView). **a)** Is a simple hairpin loop. **b)** is a two–helix bundle tile used in [10]. **c)** is two helices connected by a double crossover, analysing the flexibility of such a motif. **d)** is a possible design for a tensegrity triangle. **e)** is a six–helix bundle.

6.8 Conclusion

The oxView online tool presented has significantly simplified the setup, visualisation, and analysis of oxDNA simulations and is already helping hundreds of users every week. The software has also grown into a capable design tool on its own, with the oxDNA integration being just one of its many features. The ability to import and

combine structures from external tools also facilitates the design of modular multi–component structures such as those presented in Chapter 5. According to website analytics, the oxView tool has over 150 unique weekly users from all around the globe.

7 Discussion

There are many exciting avenues of further exploration within both polycube design and oxView development. For polycubes, one perhaps obvious comment is that there is nothing special about cubes in particular, so a more general model of self–assembling patchy shapes could prove a useful tool for many problems. The reason I choose cubes, besides being a natural 3D extension of the polyomino squares, is that they fit well in a cartesian coordinate system. However, that is a minor problem to overcome. With a more general stochastic assembler and with experimental applications such as the ever–larger polyhedral shells presented in Section 2.2.5, we might ask, given a maximum complexity - for example, a limited number of possible colours - what is the largest bounded structure that can deterministically assemble?

Another question for the future is the complexity of polycubes with staged assembly, as investigated by Demaine et al. [87]. By assembling a polycube through a hierarchy of separate stages, will an increased complexity in the number of stages decrease the number of colours and species required in an interesting way? The optimisation of other assembly parameters, such as modifying the temperature over time or colours with different interaction strengths, could also be explored for patchy particle assembly. A variable interaction matrix can also be allowed in the SAT solver specification to enable, for example, self-complementary colours.

As for the oxView project, one intriguing (but likely time–consuming) possibility would be to integrate other coarse–grained simulation options, for example, mrDNA, similarly to how oxDNA is connected using oxServe. Another ambitious feature would be to include a 2D strand view such as in caDNAno or Adenita, possibly through a connection to the also web–based scadnano.

In conclusion, the design of self–assembling nanostructures has been investigated on both an abstract and a more detailed level. The presented projects have resulted in valuable tools and methods for creating, simulating, and analysing self–limiting modular structures with minimal complexity, potentially containing building blocks created in different design software.

Appendices

A

The polycube codebase

Contents

The code used for polycube assembly can be found at `https://github.com/akodiat/polycubes`. This code repository contains both the stochastic assembly code and the SAT solver code, as detailed in the sections below.

A.1 Stochastic assembly code

The polycube assembly model was implemented in both C++ and JavaScript. The C++ implementation enables the fast command–line evaluation of input rules required for the sampling performed in Chapter 3. Meanwhile, the JavaScript implementation makes it possible to assemble and visualise polycubes in a web browser, which was useful for both model validation and for public outreach activities.

In order to speedup sampling, the c++ binary can be called multiple times in parallel and merged with the merge python script. For the results covered in Chapter 3, the sampling was done on 100 concurrent nodes and merged.

The largest sampling done, the polyomino reference with 1e9 samples (described in Section 3.3), took a total of 72 hours, 57 minutes, and 58 seconds to sample with 100 copies each performing 1e7 samples in parallel on 2.6 GHz CPUs. The merging script then took another 197 hours, 40 minutes, and 24 seconds to run, but it clearly saved a significant amount of time compared to running a single sampling for 100 times longer (which would take almost 7300 hours or 304 days).

Finally the analysis script took another 301 hours, 44 minutes, and 17 seconds to run on the same hardware.

A.1.1 C++

The stochastic assembly C++ code is found in the `cpp` directory. The Eigen and HDF5 libraries are required to compile the c++ binary. The python binding requires Eigen and the pybind library.

Install dependencies:

```
conda install -c conda-forge pybind11 eigen

wget https://www.hdfql.com/releases/2.3.0/HDFql-2.3.0_Linux64_GCC-4.9.zip
unzip HDFql-2.3.0_Linux64_GCC-4.9.zip
mv hdfql-2.3.0 ~
```

The cmake specification at `cpp/src/CMakeLists.txt` is configured to search for HDFql at the path `~/hdfql-2.3.0`, so make sure to update the file if you install another version of HDFql or if it is installed at another path.

Build the polycube binary with cmake:

```
mkdir build && cd build
cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++ ..
make
```

You should find the binary in the root of this directory (cpp)

Run ./polycubes –help for more info

A.1.2 Python binding and analysis

The C++ binary is used to sample the input rule space, as done in Chapter 3. However, to evaluate individual rules and access the stochastic assembly model through python, you can compile a pybind11 (https://github.com/pybind/pybind11) library that can be imported into python. There are build scripts and examples for both the python analysis scripts and the SAT solver code:

```
cd py
bash build_pybind.sh
```

```
cd solve/py
bash build_polycube_pybind.sh
```

This will generate a shared module `libpolycubes.so`

```
import libpolycubes as pl
pl.getCoords("040087000000")
```

The `py` directory contains a set of analysis scripts to interpret the results of the C++ binary output.

A.1.3 JavaScript

The JavaScript assembler is found in the `js` directory, with HTML files in the root of the repository. To run, you can either start a local webserver with the repository content or open the `index.html` file directly. You can also access the GitHub site at https://akodiat.github.io/polycubes.

The 3D visualisation is done with the help of the Three.js library (https://threejs.org/).

Polycube rules formatted as either decimal or hexadecimal strings (Section 3.1.1) can be supplied to the assembler as URL arguments, for example: akodiat.github.io/polycubes?decRule=|2:2|1:3||-3:0|_-1:1|-2:3||||_||||3:1| and akodiat.github.io/polycubes?rule=000a07008c00858b0000000000000000d00.

Patchy sphere preview

Use the following script to generate patchy sphere preview images

```
// Change cubes to spheres
system.changeToSpheres()

// Position camera (to get the same view angle)
camera.position.copy(new THREE.Vector3(4,4,6).add(system.centerOfMass))
camera.lookAt(system.centerOfMass)
render()

// Save image
saveCanvasImage()
```

By default, the colours of species and patches only vary in hue. They can be made more diverse by also varying their saturation and luminosity:

```
system.particleMaterials.forEach(m=>{m.color.offsetHSL(0, 0.5*(Math.random()-0.5), 0.5*(Math.random()-0.5))})
system.colorMaterials.forEach(m=>{m.color.offsetHSL(0, 0.5*(Math.random()-0.5), 0.5*(Math.random()-0.5))})
```

A.2 Polycube solver

The polycube SAT solver code is found in the `solve` directory. As for the stochastic assembly code, the SAT solver has multiple implementation. The main implementation is done in Python and can be used for production runs. The other implementation is in JavaScript and is used to draw shape specifications and to solve smaller shapes.

A.2.1 Python

To use the python solver, you need to install the python-sat library:

```
python -m pip install python-sat[pblib,aiger]
```

To verify that the solver output assembles correctly, you also need to compile the polycube python binding:

```
cd solve/py
bash build_polycube_pybind.sh
```

Finally, to solve a shape for a given number of species and colours, you call the `solve.py` script with the desired parameters:

```
python solve.py ../shapes/[shape].json [nSpecies] [nColors]
```

There are also "multi–solve" scripts to automatically create jobs solving all possible combinations of species and colour counts.

As an example of custom scripting, it is also possible to extract the CNF clauses to a file:

```
import utils
import json
from polycubeSolver import polysat

def saveClauses(nCubeTypes, nColors, solveSpecPath):
    with open(solveSpecPath, "r") as f:
        data = f.read()
    solveSpec = json.loads(data)

    mysat = polysat(
        solveSpec["bindings"],
        nCubeTypes,
        nColors,
        solveSpec["nDim"],
        solveSpec["torsion"]
    )

    name = solveSpecPath.split("/")[-1].split(".")[0]
    outPath = "{}t_{}c_{}.cnf".format(
        nCubeTypes, nColors, name
    )
    mysat.dump_cnf_to_file(outPath)

    print("Saved to {}".format(outPath))

saveClauses(6, 9, "../shapes/scaling/cube4.json")
```

A.2.2 JavaScript

The JavaScript implementation of the polycube solver is found in the `solve/js` directory, with the HTML index file at `solve/index.html` directory. To use, you can either start a local webserver with the complete repository content and navigate to `http://localhost:8080/solve/index.html`, or access the GitHub site at `https://akodiat.github.io/polycubes/solve`.

The web app provides an interface where you can interactively draw polycube shapes and save them as JSON shapes for the Polycube SAT solver. You can also use the console commands `drawFromCoords`, `drawSolidBlock`, and `drawSolidCube` to create shapes.

You can also click a button to attempt to solve the shape within the browser. This is, however, usually only feasible for small shapes using SAT, so a substitution simplifier is also employed to work towards a minimal solution by substituting similar species in the fully addressable solution.

B

Patchy particle model

As described in Section 4.5.1, the patchy particle model is used is a derivative of the oxDNA patchy particle model used by Romano et al. [40], modified to include torsional interactions to account for the polycube requirement of patch orientation alignment.

In the model, each particle is represented by a sphere covered by up to 6 patches at distance $R = 0.5$ distance units (d.u.) from the sphere's centre. The positions of the patches correspond to the three–dimensional Von Neumann neighbourhood around the origin: $\mathbf{p}_1 = R\,(0,1,0)$, $\mathbf{p}_2 = R\,(0,-1,0)$, $\mathbf{p}_3 = R\,(0,0,1)$, $\mathbf{p}_4 = R\,(0,0,-1)$, $\mathbf{p}_5 = R\,(1,0,0)$, $\mathbf{p}_6 = R\,(-1,0,0)$, Each patch is additionally assigned an orientation $\mathbf{o}$, which is equal to $\pm\mathbf{e}_1, \pm\mathbf{e}_2$, or $\pm\mathbf{e}_3$, which are the base vectors (or their negatives) of the orthonormal base of the particle. The orientation $\mathbf{o}_a$ assigned to each patch a satisfies $\mathbf{p}_a \cdot \mathbf{o}_a = 0$.

We model the interaction between patches using point–like patch interaction, where the interaction between two patches a and b on two distinct particles i and j is given by the following interaction potential:

$$V_{\text{patch}}(\mathbf{r}_{ij}, \boldsymbol{\Omega}_{ij}) = \delta_{ab} V_{\text{pdist}}(r_p) V_{\text{angle}}(\theta_a, \theta_b, \theta_t), \tag{B.1}$$

where δ_{ab} is 1 if colours assigned to patch a and b are compatible and 0 otherwise. The patch potential consists of two components: potential V_{pdist} that only depends

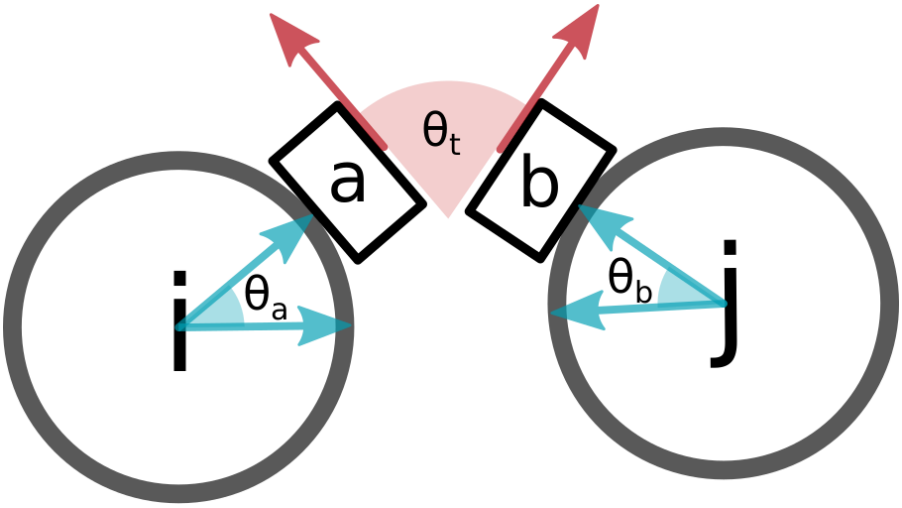


Figure B.1: Schematic of the patchy particle alignment angles. Note that, while the figure is drawn in 2D, the particles and vectors are in fact three–dimensional and are not restricted to the depicted plane. The angle θ_a is measured between the vectors $\hat{\mathbf{r}}_{\mathbf{ij}}$ (pointing from particle i to particle j) and $\hat{\mathbf{p}}_{\mathbf{a}}$ (pointing from from particle i to its patch a. θ_b. Likewise, the angle θ_b is measured between the vectors $-\hat{\mathbf{r}}_{\mathbf{ij}}$ (pointing from particle j to particle i) and $\hat{\mathbf{p}}_{\mathbf{b}}$ (pointing from from particle j to its patch n. Finally the angle θ_t is measured between the orientation vectors of the two patches, $\mathbf{o}_a$ and $\mathbf{o}_b$, which are always orthogonal to $\hat{\mathbf{p}}_{\mathbf{a}}$ and $\hat{\mathbf{p}}_{\mathbf{b}}$ respectively.

on distance between the two patches r_{ab}, and $V_{\text{angle}}(\theta_a, \theta_b, \theta_t)$ which depends on the mutual orientation $\mathbf{\Omega}_{ij}$ of the two particles as given by angles that are calculated as follows:

$$\cos\theta_a = \hat{\mathbf{r}}_{ij} \cdot \hat{\mathbf{p}}_a, \quad \cos\theta_b = -\hat{\mathbf{r}}_{ij} \cdot \hat{\mathbf{p}}_b, \quad \cos\theta_t = \mathbf{o}_a \cdot \mathbf{o}_b, \tag{B.2}$$

where the above vectors are normalised to 1, as indicated by the hat symbol. As seen in Figure B.1, angles θ_a and θ_b correspond to the angle between the normalised vector between the centres of mass of patchy particle i and j and the normalised vector pointing to patch a or b respectively. Angle θ_t corresponds to the angle between the orientations associated with the respective patches. The interaction potential is

$$V_{\text{angle}}(\theta_a, \theta_b, \theta_t) = V_{\text{angmod}}(\theta_a) V_{\text{agmod}}(\theta_b) V_{\text{agmod}}(\theta_t), \tag{B.3}$$

where V_{agmod} is angular modulation function defined to be equal to one if the angle θ equals to the desired angle θ_0 (which are set to 0, requiring perfect alignment of the

respective vectors), and parameters a and Δ then define the width of the potential.

$$V_{\text{angmod}}(\theta) = \begin{cases} V_{\text{mod}}(\theta, a, \theta_0) & \text{if } \theta_0 - \Delta < \theta < \theta_0 + \Delta, \\ V_{\text{smooth}}(\theta, b, \theta_0 - \Delta_c) & \text{if } \theta_0 - \Delta_c < \theta < \theta_0 - \Delta, \\ V_{\text{smooth}}(\theta, b, \theta_0 + \Delta_c) & \text{if } \theta_0 + \Delta < \theta < \theta_0 + \Delta_c, \\ 0 & \text{otherwise.} \end{cases} \tag{B.4}$$

where the additional parameters b and Δ_c are set so that the piece–wise function $V_{\text{angmod}}(\theta)$ is differentiable (See Figure B.2). The potentials used in the definition are

$$V_{\text{smooth}}(x, b, x_c) = b(x_c - x)^2, \tag{B.5}$$

and

$$V_{\text{mod}}(\theta, a, \theta_0) = 1 - a(\theta - \theta_0)^2. \tag{B.6}$$

Unless otherwise specified, the default narrow type 0 was used, with $a = 0.46$ and $\Delta = 0.7$ for V_{angmod} as seen in Figure B.2.

The additional distance–modulation term in the potential between a pair of patches on two distinct particles is

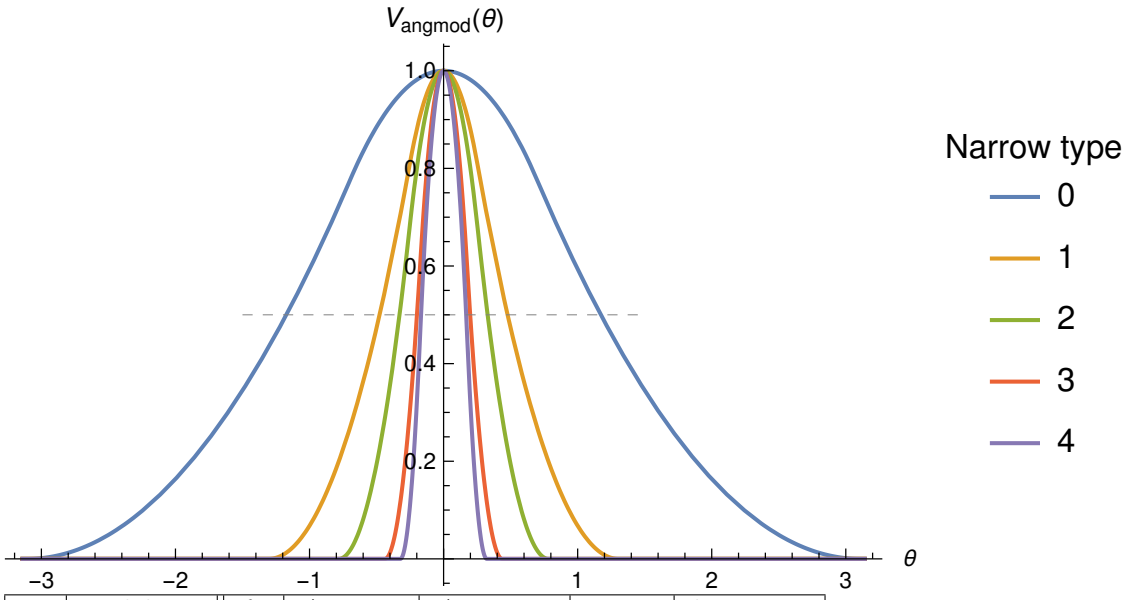
$$V_{\text{pdist}}(r_p) = \begin{cases} -1.001 \exp\left[-\left(\frac{r_p}{\alpha}\right)^{10}\right] + C & \text{if } r_p \leq r_{\text{pmax}} \\ 0 & \text{otherwise} \end{cases} \tag{B.7}$$

where r_p is the distances between a pair of patches, and $\alpha = 0.12\,\text{d.u.}$ sets the patch width. The constant C is set so that $V_{\text{patch}}(r_{\text{pmax}}) = 0$ for $r_{\text{pmax}} = 0.18\,\text{d.u.}$. The patchy particles further interact through excluded volume interactions ensuring that two particles do not overlap:

$$f_{\text{exc}}(r, \epsilon, \sigma, r^\star) = \begin{cases} V_{\text{LJ}}(r, \epsilon, \sigma) & \text{if } r < r^\star, \\ \epsilon V_{\text{smooth}}(r, b, r^c) & \text{if } r^\star < r < r^c, \\ 0 & \text{otherwise.} \end{cases} \tag{B.8}$$

where r is the distance between the centers of mass of the patchy particles, and σ is set to $2R = 1.0$ distance units, twice the desired radius of the patchy particle. The repulsive potential is a piecewise function consisting of the Lennard–Jones potential function:

$$V_{\text{LJ}}(r, \sigma) = 8\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]. \tag{B.9}$$



nt	width	θ_0	Δ	Δ_c	a	b
0	2.345750	0	0.7	3.105590	0.46	0.133855
1	0.954736	0	0.2555	1.304631	3	0.730604
2	0.656996	0	0.2555	0.782779	5	2.42282
3	0.396613	0	0.17555	0.438183	13	8.689492
4	0.336622	0	0.17555	0.322741	17.65	21.0506

Figure B.2: Patchy particle narrow types, causing patches of different widths. The table shows the constants used for each narrow type, with the plot showing the different angular modulation potentials as a function of angle in radians. The width is measured between the two points intersecting the dashed line, where $V_{\text{angmod}} = \frac{1}{2}$. The default narrow type 0 is the least narrow.

that is truncated using a quadratic smoothing function from (B.5), with b and x_c set so that the potential is a differentiable function that is equal to 0 after a specified cutoff distance $r^c = 0.8$.

The patchy particle system was simulated using rigid–body Molecular Dynamics with an uring the simulation, each patch was only able to bind to one other patch at the time, and if the binding energy between a pair of patches, as given by Eq. (B.7), is smaller than 0, none of the patches can bind to any other patch until their pair interaction potential is again 0.

B.1 Patchy particle code

The modified version of oxDNA used to simulate torsional patchy particles can be found at `https://github.com/Akodiat/oxDNA_torsion`. To download and combile, run:

```
git clone https://github.com/Akodiat/oxDNA_torsion.git
cd oxDNA_torsion
mkdir build
cd build
cmake ..
make -j4
make romano
```

Simulation files for a given polycube shape can then be generated through the web console at `https://akodiat.github.io/polycubes`, by running:

```
getPatchySimFiles(
  "0700000705008687000000000", // Rule string
  1, // Number of assemblies
  "cube", // Name
  "/users/joakim/repo/oxDNA_torsion", // Path to oxDNA directory
  [.01,.02,.03,.04,.05,.06,.07,.08,.09,.1], // Temperature range
  0.1 // Density (used to generate configuration when the number
  // of assemblies is more than one)
)
```

If the number of assemblies is one, a configuration file will be generated specifying the assembled shape. Else, the `generateConf.sh` script can be used to generate a random configuration with the provided density.

C

The oxView codebase

The code for oxView can be found at `https://github.com/sulcgroup/oxdna-viewer`. It is written in TypeScript and compiled into JavaScript. The 3D visualisation is done with the help of the Three.js library (`https://threejs.org/`), while the graphical user interface uses the Metro 4 library (`https://metroui.org.ua/`).

If you make any changes, you need both TypeScript (`https://www.typescriptlang.org/`) and Node.js (`https://nodejs.org/`) installed to compile. Typing `npm install` in the source directory should install all required node modules. Then type `tsc` to compile the typescript into javascript.

More detailed documentation can be found in the README.md file in the repository: `https://github.com/sulcgroup/oxdna-viewer/blob/master/README.md`

References

1. The DNA-Robotics Innovative Training Network. *Research - DNA-Robotics* Nov. 2017. https://dna-robotics.eu/research/.
2. The DNA-Robotics Innovative Training Network. *UOXF: Modular assembly of integrated nanorobotic systems* Dec. 2017. https://dna-robotics.eu/job-positions/uoxf-modular-assembly-of-integrated-nanorobotic-systems/.
3. Seeman, N. C. *Structural DNA Nanotechnology* (Cambridge University Press, 2016).
4. Calladine, C. R. & Drew, H. *Understanding DNA: the molecule and how it works* (Academic press, 1997).
5. Seeman, N. C. Nucleic Acid Junctions and Lattices. *Journal of Theoretical Biology* **99,** 237–247 (1982).
6. Rothemund, P. W. Folding DNA to create nanoscale shapes and patterns. *Nature* **440,** 297 (2006).
7. Dey, S. *et al.* DNA origami. *Nature Reviews Methods Primers* **1,** 1–24 (2021).
8. Sadava, D. E., Hillis, D. M., Heller, H. C. & Berenbaum, M. in. 10th ed. Chap. 14 (Macmillan, 2014).
9. Guo, P. The emerging field of RNA nanotechnology. *Nature nanotechnology* **5,** 833 (2010).
10. Geary, C., Rothemund, P. W. & Andersen, E. S. A single-stranded architecture for cotranscriptional folding of RNA nanostructures. *Science* **345,** 799–804 (2014).
11. Sparvath, S. L., Geary, C. W. & Andersen, E. S. in *3D DNA Nanostructure* 51–80 (Springer, 2017).
12. Geary, C., Grossi, G., McRae, E. K., Rothemund, P. W. & Andersen, E. S. RNA origami design tools enable cotranscriptional folding of kilobase-sized nanoscaffolds. *Nature chemistry* **13,** 549–558 (2021).
13. Seeman, N. C. Nucleic acid junctions and lattices. *Journal of theoretical biology* **99,** 237–247 (1982).
14. Winfree, E. *Algorithmic self-assembly of DNA* PhD thesis (California Institute of Technology, 1998).
15. Winfree, E., Liu, F., Wenzler, L. A. & Seeman, N. C. Design and self-assembly of two-dimensional DNA crystals. *Nature* **394,** 539 (1998).
16. Ong, L. L. *et al.* Programmable self-assembly of three-dimensional nanostructures from 10,000 unique components. *Nature* **552,** 72–77 (2017).
17. Tikhomirov, G., Petersen, P. & Qian, L. Fractal assembly of micrometre-scale DNA origami arrays with arbitrary patterns. *Nature* **552,** 67 (2017).

18. Tikhomirov, G., Petersen, P. & Qian, L. Programmable disorder in random DNA tilings. *Nature nanotechnology* **12,** 251–259 (2017).

19. Wagenbauer, K. F., Sigl, C. & Dietz, H. Gigadalton-scale shape-programmable DNA assemblies. *Nature* **552,** 78 (2017).

20. Woo, S. & Rothemund, P. W. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature chemistry* **3,** 620–627 (2011).

21. Sigl, C. *et al.* Programmable icosahedral shell system for virus trapping. *Nature Materials* **20,** 1281–1289 (2021).

22. Lin, Z. *et al.* Engineering Organization of DNA Nano-Chambers through Dimensionally Controlled and Multi-Sequence Encoded Differentiated Bonds. *Journal of the American Chemical Society* **142.** PMID: 32902966, 17531–17542 (2020).

23. Wang, M. *et al.* Programmable Assembly of Nano-architectures through Designing Anisotropic DNA Origami Patches. *Angewandte Chemie International Edition* **59,** 6389–6396. https://onlinelibrary.wiley.com/doi/abs/10.1002/anie.201913958 (2020).

24. Wang, H. Proving theorems by pattern recognition—II. *Bell system technical journal* **40,** 1–41 (1961).

25. Doty, D. Theory of algorithmic self-assembly. *Communications of the ACM* **55,** 78–88 (2012).

26. Doty, D. *DNA tile self-assembly* Tutorial presented at the 23rd International Conference on DNA Computing and Molecular Programming. Sept. 2017. https://web.cs.ucdavis.edu/~doty/papers/dna23-tile-assembly-tutorial.pdf.

27. Ahnert, S., Johnston, I., Fink, T., Doye, J. & Louis, A. Self-assembly, modularity, and physical complexity. *Physical Review E* **82,** 026117 (2010).

28. Johnston, I. G., Ahnert, S. E., Doye, J. P. & Louis, A. A. Evolutionary dynamics in a simple model of self-assembly. *Physical Review E* **83,** 066105 (2011).

29. Li, M. & Vitányi, P. M. B. *An introduction to Kolmogorov complexity and its applications [electronic resource]* Fourth edition. eng. ISBN: 9783030112981 (electronic book) (Cham, 2019).

30. Dingle, K., Camargo, C. Q. & Louis, A. A. Input–output maps are strongly biased towards simple outputs. *Nature communications* **9,** 761 (2018).

31. Dingle, K., Pérez, G. V. & Louis, A. A. Generic predictions of output probability based on complexities of inputs and outputs. *Scientific reports* **10,** 1–9 (2020).

32. Johnston, I. G. *et al.* Symmetry and simplicity spontaneously emerge from the algorithmic nature of evolution. *Proceedings of the National Academy of Sciences* **119,** e2113883119 (2022).

33. Lempel, A. & Ziv, J. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory* **22,** 75–81 (1976).

34. Sloane, N. & Plouffe, S. *The encyclopedia of integer sequences* (1995).

35. Sloane, N. & Redelmeier, D. H. *A000988 - Number of one-sided polyominoes with n cells* https://oeis.org/A000988 (2020).

36. Romano, F., Russo, J., Kroc, L. & Šulc, P. Designing patchy interactions to self-assemble arbitrary structures. *Physical Review Letters* **125,** 118003 (2020).

37. Audemard, G. & Simon, L. Glucose: a solver that predicts learnt clauses quality. *SAT Competition,* 7–8 (2009).

38. Ignatiev, A., Morgado, A. & Marques-Silva, J. *PySAT: A Python Toolkit for Prototyping with SAT Oracles* in *SAT* (2018), 428–437. https://doi.org/10.1007/978-3-319-94144-8_26.

39. Russo, J., Tartaglia, P. & Sciortino, F. Reversible gels of patchy particles: role of the valence. *The Journal of chemical physics* **131,** 014504 (2009).

40. Rovigatti, L., Šulc, P., Reguly, I. Z. & Romano, F. A comparison between parallelization approaches in molecular dynamics simulations on GPUs. *Journal of computational chemistry* **36,** 1–8 (2015).

41. Hagberg, A. A., Schult, D. A. & Swart, P. J. *Exploring Network Structure, Dynamics, and Function using NetworkX* in *Proceedings of the 7th Python in Science Conference* (eds Varoquaux, G., Vaught, T. & Millman, J.) (Pasadena, CA USA, 2008), 11–15.

42. Douglas, S. M. *et al.* Rapid prototyping of 3D DNA-origami shapes with caDNAno. *Nucleic Acids Research* **37,** 5001–5006. ISSN: 03051048 (2009).

43. Doty, D., Lee, B. L. & Stérin, T. *scadnano: A browser-based, scriptable tool for designing DNA nanostructures* in *DNA 2020: Proceedings of the 26th International Meeting on DNA Computing and Molecular Programming* (eds Geary, C. & Patitz, M. J.) **174** (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020), 9:1–9:17. ISBN: 978-3-95977-163-4. https://drops.dagstuhl.de/opus/volltexte/2020/12962.

44. Autodesk, INC. *Maya* version 2015. https:/autodesk.com/maya.

45. Conway, N. & Douglas, S. *Windows Installation - caDNAno* https://cadnano.org/windows-installation.html (2021).

46. Benson, E. *et al.* DNA rendering of polyhedral meshes at the nanoscale. *Nature* **523,** 441–444. ISSN: 14764687 (2015).

47. Benson, E. *et al.* Computer-Aided Production of Scaffolded DNA Nanostructures from Flat Sheet Meshes. *Angewandte Chemie International Edition* **55,** 8869–8872 (2016).

48. Benson, E. *vHelix - Free-form DNA-nanostructure design* http://www.vhelix.net/ (2021).

49. Jun, H. *et al.* Rapid prototyping of arbitrary 2D and 3D wireframe DNA origami. *Nucleic Acids Research.* gkab762. ISSN: 0305-1048. https://doi.org/10.1093/nar/gkab762 (Sept. 2021).

50. Williams, S. *et al. Tiamat: A Three-Dimensional Editing Tool for Complex DNA Structures* in *DNA Computing* (eds Goel, A., Simmel, F. C. & Sosík, P.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009), 90–101. ISBN: 978-3-642-03076-5.

51. Yan, H. *Tiamat download* http://yanlab.asu.edu/Resources.html (2021).

52. Miao, H. *et al.* Multiscale Visualization and Scale-adaptive Modification of DNA Nanostructures. *IEEE Transactions on Visualization and Computer Graphics* **24.** https://www.cg.tuwien.ac.at/research/publications/2018/miao_tvcg_2018/ (Jan. 2018).

53. OneAngstrom. *SAMSON connect* version SAMSON 0.7.0. https://www.samson-connect.net/.

54. Huang, C.-M., Kucinic, A., Johnson, J. A., Su, H.-J. & Castro, C. E. Integrated computer-aided engineering and design for DNA assemblies. *Nature Materials,* 1–8 (2021).

55. Poppleton, E. *et al.* Design, optimization and analysis of large DNA and RNA nanostructures through interactive visualization, editing and molecular simulation. *Nucleic acids research* **48,** e72–e72 (2020).

56. Bohlin, J. *et al.* Design and simulation of DNA, RNA and hybrid protein–nucleic acid nanostructures with oxView. *Nature Protocols.* ISSN: 1750-2799. https://doi.org/10.1038/s41596-022-00688-5 (June 2022).

57. Phillips, J. C. *et al.* Scalable molecular dynamics with NAMD. *Journal of computational chemistry* **26,** 1781–1802 (2005).

58. Cornell, W. D. *et al.* A second generation force field for the simulation of proteins, nucleic acids, and organic molecules J. Am. Chem. Soc. 1995, 117, 5179- 5197. *Journal of the American Chemical Society* **118,** 2309–2309 (1996).

59. Brooks, B. R. *et al.* CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. *Journal of computational chemistry* **4,** 187–217 (1983).

60. Yoo, J. & Aksimentiev, A. In situ structure and dynamics of DNA origami determined through molecular dynamics simulations. *Proceedings of the National Academy of Sciences* **110,** 20099–20104 (2013).

61. Sengar, A., Ouldridge, T. E., Henrich, O., Rovigatti, L. & Sulc, P. A primer on the oxDNA model of DNA: When to use it, how to simulate it and how to interpret the results. *arXiv preprint arXiv:2104.11567* (2021).

62. Ouldridge, T. E., Louis, A. A. & Doye, J. P. DNA nanotweezers studied with a coarse-grained model of DNA. *Physical review letters* **104,** 178101 (2010).

63. SantaLucia, J. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. *Proceedings of the National Academy of Sciences* **95,** 1460–1465 (1998).

64. Sharma, R., Schreck, J. S., Romano, F., Louis, A. A. & Doye, J. P. Characterizing the motion of jointed DNA nanostructures using a coarse-grained model. *ACS nano* **11,** 12426–12435 (2017).

65. Doye, J. P. K. *et al.* The oxDNA coarse-grained model as a tool to simulate DNA origami. *Methods in Molecular Biology, submitted.* https://arxiv.org/abs/2004.05052 (2020).

66. Ouldridge, T. E., Louis, A. A. & Doye, J. P. Structural, mechanical, and thermodynamic properties of a coarse-grained DNA model. *The Journal of chemical physics* **134,** 02B627 (2011).

67. Šulc, P. *et al.* Sequence-dependent thermodynamics of a coarse-grained DNA model. *The Journal of Chemical Physics* **137,** 135101. https://doi.org/10.1063/1.4754132 (2012).

68. Ouldridge, T. E. *et al.* Optimizing DNA nanotechnology through coarse-grained modeling: a two-footed DNA walker. *ACS nano* **7,** 2479–2490 (2013).

69. Snodin, B. E. *et al.* Introducing improved structural properties and salt dependence into a coarse-grained model of DNA. *The Journal of chemical physics* **142,** 06B613_1 (2015).

70. Šulc, P., Romano, F., Ouldridge, T. E., Doye, J. P. & Louis, A. A. A nucleotide-level coarse-grained model of RNA. *The Journal of chemical physics* **140,** 06B614_1 (2014).

71. Procyk, J., Poppleton, E. & Šulc, P. Coarse-grained nucleic acid–protein model for hybrid nanotechnology. *Soft Matter* **17,** 3586–3593 (2021).

72. Poppleton, E., Romero, R., Mallya, A., Rovigatti, L. & Šulc, P. OxDNA.org: a public webserver for coarse-grained simulations of DNA and RNA nanostructures. *Nucleic Acids Research* **49,** W491–W498. ISSN: 0305-1048. https://doi.org/10.1093/nar/gkab324 (May 2021).

73. Maffeo, C. & Aksimentiev, A. MrDNA: a multi-resolution model for predicting the structure and dynamics of DNA systems. *Nucleic Acids Research* (2020).

74. Gerling, T., Wagenbauer, K. F., Neuner, A. M. & Dietz, H. Dynamic DNA devices and assemblies formed by shape-complementary, non–base pairing 3D components. *Science* **347,** 1446–1452 (2015).

75. Zadegan, R. M. *et al.* Construction of a 4 zeptoliters switchable 3D DNA box origami. *ACS nano* **6,** 10050–10053 (2012).

76. Liedl, T., Högberg, B., Tytell, J., Ingber, D. E. & Shih, W. M. Self-assembly of three-dimensional prestressed tensegrity structures from DNA. *Nature nanotechnology* **5,** 520 (2010).

77. Han, D., Pal, S., Liu, Y. & Yan, H. Folding and cutting DNA into reconfigurable topological nanostructures. *Nature nanotechnology* **5,** 712 (2010).

78. Castro, C. E. *et al.* A primer to scaffolded DNA origami. *Nature methods* **8,** 221 (2011).

79. Kim, D.-N., Kilchherr, F., Dietz, H. & Bathe, M. Quantitative prediction of 3D solution shape and flexibility of nucleic acid nanostructures. *Nucleic acids research* **40,** 2862–2868 (2012).

80. Suma, A. *et al.* TacoxDNA: A user-friendly web server for simulations of complex DNA structures, from single strands to origami. *Journal of Computational Chemistry.* https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.26029 (2019).

81. Suma, A. *et al.* TacoxDNA: A user-friendly web server for simulations of complex DNA structures, from single strands to origami. *Journal of computational chemistry* **40,** 2586–2595 (2019).

82. Benson, E., Carrascosa Marzo, R., Bath, J. & Turberfield, A. J. Strategies for Constructing and Operating DNA Origami Linear Actuators. *Small* **17,** 2007704 (2021).

83. Baraff, D. *An introduction to physically based modeling: Rigid Body Simulation I — Unconstrained Rigid Body Dynamics* 1997. http://www.cs.cmu.edu/~baraff/pbm/rigid1.pdf.

84. Ester, M., Kriegel, H.-P., Sander, J., Xu, X., *et al. A density-based algorithm for discovering clusters in large spatial databases with noise.* in *Kdd* **96** (1996), 226–231.

85. Goodman, R. P. *et al.* Rapid chiral assembly of rigid DNA building blocks for molecular nanofabrication. *Science* **310,** 1661–1665 (2005).

86. Bai, L. & Breen, D. Calculating Center of Mass in an Unbounded 2D Environment. *Journal of Graphics Tools* **13,** 53–60. https://doi.org/10.1080/2151237X.2008.10129266 (2008).

87. Demaine, E. D. *et al.* Staged self-assembly: nanomanufacture of arbitrary shapes with O (1) glues. *Natural Computing* **7,** 347–370 (2008).