View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

scadnano: A Browser-Based, Scriptable Tool for Designing DNA Nanostructures

David Doty[1]
University of California, Davis, CA, USA
https://web.cs.ucdavis.edu/~doty/
doty@ucdavis.edu

Benjamin L Lee
University of California, Davis, CA, USA
bnllee@ucdavis.edu

Tristan Stérin
Maynooth University, Ireland
https://dna.hamilton.ie/tsterin/index.html
Tristan.Sterin@mu.ie

Abstract

We introduce *scadnano* (short for "scriptable cadnano"), a computational tool for designing synthetic DNA structures. Its design is based heavily on cadnano [24], the most widely-used software for designing DNA origami [33], with three main differences:

1. scadnano runs entirely in the browser, with *no software installation* required.
2. scadnano designs, while they can be edited manually, can also be created and edited by a *well-documented Python scripting library*, to help automate tedious tasks.
3. The scadnano file format is *easily human-readable*. This goal is closely aligned with the scripting library, intended to be helpful when debugging scripts or interfacing with other software. The format is also somewhat more *expressive* than that of cadnano, able to describe a broader range of DNA structures than just DNA origami.

2012 ACM Subject Classification Applied computing → Computer-aided design

Keywords and phrases computer-aided design, structural DNA nanotechnology, DNA origami

Digital Object Identifier 10.4230/LIPIcs.DNA.2020.9

Supplementary Material *stable/dev apps:* https://scadnano.org, https://scadnano.org/dev
repositories: https://github.com/UC-Davis-molecular-computing/scadnano
https://github.com/UC-Davis-molecular-computing/scadnano-python-package
Python library API: https://scadnano-python-package.readthedocs.io
tutorials: https://github.com/UC-Davis-molecular-computing/scadnano-python-package/blob/master/tutorial/tutorial.md, https://github.com/UC-Davis-molecular-computing/scadnano/blob/master/tutorial/tutorial.md

Funding *David Doty*: Supported by NSF grants 1619343, 1900931, and CAREER grant 1844976.
Benjamin L Lee: Supported by REU supplement through NSF CAREER grant 1844976.
Tristan Stérin: Supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772766, Active-DNA project), and Science Foundation Ireland (SFI) under Grant number 18/ERCS/5746.

Acknowledgements We thank Matthew Patitz for beta-testing and feedback, and Pierre-Étienne Meunier, author of codenano, for valuable discussions regarding the data model/file format. We are grateful to anonymous reviewers whose detailed feedback has increased the presentation quality.

[1] Corresponding author

© David Doty, Benjamin L Lee, and Tristan Stérin;
licensed under Creative Commons License CC-BY
26th International Conference on DNA Computing and Molecular Programming (DNA 26).
Editors: Cody Geary and Matthew J. Patitz; Article No. 9; pp. 9:1–9:17

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 DNA origami and cadnano

Since its inception almost 15 years ago, DNA origami [33] has stood as the most reliable, high-yield, and low-cost method for synthesizing uniquely addressed DNA nanostructures, on the order of 100 nm wide, with ≈ 6 nm addressing resolution (i.e., that's how far apart individual strands are).[2] To create the original designs, Rothemund wrote custom Matlab scripts to generate and visualize the designs (with ASCII art). Soon after, the software cadnano was developed by Douglas et al. [24], as part of a project extending the original 2D DNA origami results to 3D structures [23]. cadnano has become a standard tool in structural DNA nanotechnology, used for describing most major DNA origami designs.

1.2 scadnano

The scadnano graphical interface is shown in Figure 1; it mimics that of cadnano.

The goal of scadnano is to aid in designing large-scale DNA nanostructures, such as DNA origami, with ability to edit structures either manually, or programmatically through a scripting library. scadnano seeks to imitate most of the features of cadnano, with three major differences that enhance the usability and interoperability of scadnano:

1. scadnano runs entirely in the browser, with *no software installation* required. It aims, above all else, to be simple and easy to use, well-suited for teaching, for example.
2. scadnano designs, while they can be edited manually, can also be created and edited by a *well-documented Python scripting library*, to help automate tedious tasks.[3]
3. The scadnano file format is *easily human-readable* and expressive, natural for describing a broader range of DNA structures than just DNA origami. This goal is closely aligned with the scripting library, useful when debugging scripts or interfacing with other software. A related project, codenano [5], uses essentially the same file format, developed simultaneously in consultation with the main author of codenano.

The major features of scadnano are described in more detail in Section 3. Designed with interoperability in mind, any cadnano design can be imported into scadnano, and scadnano designs obeying certain constraints (see Section 2.3) can be exported to cadnano.

1.3 Related work

cadnano is the most related prior work, and its design was the inspiration for scadnano. Section 3.1 goes into detail about features that scadnano shares in common with cadnano, and the rest of Section 3 discusses some extra features in scadnano. codenano is close in purpose to scadnano [5], being also browser-based and scriptable. Unlike scadnano, codenano includes 3D visualisation components but not graphical editing.

[2] The basic idea of DNA origami is to use a long *scaffold* strand (either synthesized or natural; the most common choice is the natural circular single-stranded virus known as M13mp18, 7249 bases long), and to synthesize shorter (a few dozen bases long) *staple* strands designed to bind to multiple regions of the scaffold. Upon mixing in standard DNA self-assembly buffer conditions (e.g., 10 mM Tris, 1 mM EDTA, pH 8.0, 12.5 mM $MgCl_2$), with staples "significantly" more concentrated than the scaffold (typical concentrations are 1 nM scaffold and 10 nM each staple), and annealing from 90°C to 20°C for one hour, the staples bind to the scaffold and fold it into the desired shape, while excess staples remain free in solution and are easily separated from the formed structures by standard purification techniques.

[3] cadnano v2.5 has a Python scripting library, but its documentation is incomplete [3], and cadnano v2.5 has not been updated for two years [2] at the time of this writing.

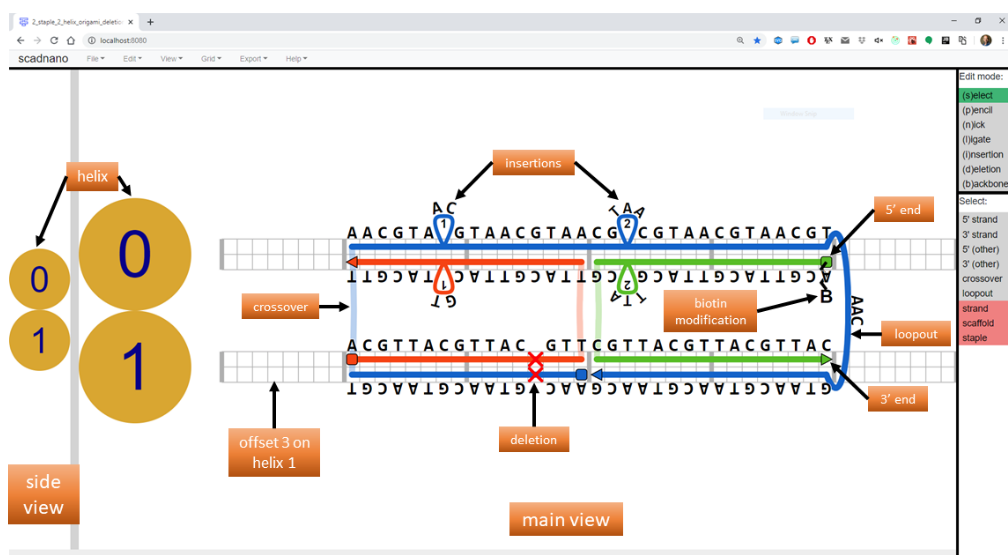


Figure 1 screenshot of scadnano, annotated with some labels (in orange rectangles) to point out various parts of the data model.[4] The center part is the *main view*, which shows the x and y coordinates; most editing takes place here. On the left is the *side view*, which shows the z and y coordinates. y increases going *down* in both views (so-called "screen coordinates), x increases going *right* in the main view and going *into* the screen in the side view. z increases going *right* in the side view and going *out of* the screen in the main view. The Edit modes on the right change what sorts of edits are possible, and the Select modes change what sort of objects can be selected while in the "select" edit mode.

vHelix [18] offers comprehensive 3D origami editing and visualisation features but relies on Autodesk Maya. Adenita [21] is a design and visualisation tool that allows one to work with various DNA nanostructures: standard parallel-helix DNA origami, wireframe origamis [28], and tile-based designs. Adenita is distributed within the SAMSON [17] molecular modeling platform. Specific to the domain of 2D and 3D wireframe origamis, ATHENA [28] provides both an editing interface and sequence design algorithms that generate staple sequences from a 2D sketch. Not related to graphical or script-based DNA design editing, the following software provides structural prediction tools for various features of DNA designs: CanDo [4] (finite elements-based 3D structure prediction), NUPACK and ViennaRNA [30,43] (thermodynamic energy of DNA strands), oxDNA [38] (kinetics prediction by molecular dynamics simulation), and MrDNA [31] (3D structure and kinetics prediction).

1.4 Paper outline

Section 2 describes the data model used by scadnano to represent a DNA design, and its closely related storage file format, including a comparison with cadnano's file format. Section 3 describes several features of scadnano, including some that are absent from cadnano. Section 4 explains the software architecture of scadnano. Section 4 is not necessary to understand how to use scadnano, but it helps to justify why scadnano may be simpler to maintain and enhance in the future. Section 5 discusses possible future features.

This paper is not a self-contained document describing scadnano in full. See the supplementary material links for online documentation, tutorials, and the Python library API.

[4] This design is intended merely to show some scadnano features, not to show proper design respecting DNA crossover geometry; it would be strained if actually assembled.

2 Data model and file format

2.1 scadnano data model

Although scadnano and its data model are natural for describing DNA origami, it can be used to describe any DNA nanostructure composed of several DNA strands. Like cadnano, scadnano is especially well-suited to structures where all DNA helices are parallel, which includes not only origami, but also certain tile-based designs (e.g., [39, 40, 42]), or "criss-cross slat" assembly [32]. The basic concepts, explained in more detail below, are that the design is composed of several *strands*, which are bound to each other on some domains, and possibly single-stranded on others, and double-stranded portions of DNA occupy a *helix*.

DNA Design

An example DNA design is shown in Figure 1, showing most of the features discussed here. A design (the type of object stored in a `.sc` file produced when clicking "Save" in scadnano) consists of a `grid` type (a.k.a., *lattice*, one of the following types: `square`, `honeycomb`, `hex`, or `none`, explained below), a list of `helices`, and a list of `strands`. The order of strands in the list generally doesn't matter, although it influences which are drawn on top, so a strand later in the list will have its crossovers drawn over the top of earlier strands.

Helices

Unlike strands, the order of the helices matters; if there are h helices, the helices are numbered 0 through $h-1$. This can be overridden by specifying a field called `idx` in each helix, but the default is to number them consecutively. Each helix defines a set of integer *offsets* with a minimum and maximum; in the example above, the minimum and maximum for each helix are 0 and 48, respectively, so 48 total offsets are shown. Each offset is a position where a DNA base of a strand can go.

Helices in a grid (meaning one of square, honeycomb, or hex) have a 2D integer `grid_position` depicted in the side view (see Figure 3). Helices without a grid (meaning grid type none) have a `position`, a 3D real vector describing their `x`, `y`, `z` coordinates. Each Helix also has fields to describe angular orientation, using the "aircraft principle axes" `pitch`, `roll`, and `yaw` (default 0), although this feature is currently not well-supported (`https://github.com/UC-Davis-molecular-computing/scadnano/issues/39`). The coordinates of helices in the main view depends on `grid_position` if a grid is used, and on `position` otherwise. (Each grid position is essentially interpreted as a position with `z` = `pitch` = `roll` = `yaw` = 0.) Helices are listed from top to bottom in the order they appear in the sequence, unless the property `helices_view_order` is specified in the design to display them in a different order, though currently this can only be done in the scripting library.

Helix.`roll` describes the DNA backbone rotation about the long axis of the helix. At the offset `Helix.min_offset`, the backbone of the forward strand on that helix has angle `Helix.roll`, where we define 0 degrees to point to straight *up* in the side view. Rotation is *clockwise* as the rotation increases from 0 up to 360 degrees. This feature is not intended as a globally predictive model of stability. Rather, it helps visualize backbone angles, to place crossovers that minimize strain, by ensuring crossovers are "locally consistent", without enforcing a global notion of absolute backbone rotation on all offsets in the system.

Strands and domains

Each strand is defined primarily by an ordered list of `domains`. Each domain is either a single-stranded *loopout* not associated to any helix, or it is a *bound domain*: a region of the strand that is contiguous on a single helix. The phrase is a bit misleading, since a bound domain is not necessarily bound to another strand, but the intention is for most of them to be bound, and for single-stranded regions usually to be represented by loopouts.

Each bound domain is specified by four mandatory properties: `helix` (indicating the index of the helix on which the domain resides), `forward` (a direction can be forward or reverse, indicated by whether this field is true or false), `start` integer offset, and a larger `end` integer offset. As with common string/list indexing in programming languages, `start` is inclusive but `end` is exclusive. So for example, a bound domain with `end`=8 is adjacent to one with `start`=8. In the main view, `forward` bound domains are depicted on the top half of the helix, and *reverse* (those with `forward`=false) are on the bottom half. If a bound domain is forward, then `start` is the offset of its 5' end, and `end`−1 is the offset of its 3' end, otherwise these roles are reversed. There is implicitly a crossover between adjacent bound domains in a strand. Loopouts are explicitly specified as a (non-bound) domain in between two bound domains. Currently, two loopouts cannot be consecutive (and this will remain a requirement), and a loopout cannot be the first or last domain of a strand (this may be relaxed in the future).

Bound domains may have optional fields, notably `deletions` (called *skips* in cadnano) and `insertions` (called *loops* in cadnano). They are a visual trick used to allow bound domains to appear to be one length in the main view of scadnano, while actually having a different length. Normally, each offset represents a single base. If instead a deletion appears at that offset, then it does not correspond to any DNA base. If an insertion appears at that offset, it has a positive integer `length`: the number of bases represented by that offset is `length`+1.

Strand optional fields

Each strand also has a `color` and a Boolean field `is_scaffold`. DNA origami designs have at least one strand that is a scaffold (but can have more), and a non-DNA-origami design is simply one in which every strand has `is_scaffold` = false. Unlike cadnano, a scaffold strand can have either direction on any helix. When there is at least one scaffold, all non-scaffold strands are called *staples*. The general idea behind DNA origami is that all binding is between scaffolds and staples, never scaffold-scaffold or staple-staple. However, this convention is not enforced by scadnano; there are legitimate reasons for non-scaffold strands to bind to each other (e.g., DNA walkers [26] or circuits [20] on the surface of an origami).

A strand can have an optional DNA `sequence`. Of course, since the whole point of this software is to help design DNA structures, at some point a DNA sequence should be assigned to some of the strands. However, it is often best to mostly finalize the design before assigning a DNA sequence, which is why the field is optional. Many of the operations attempt to keep things consistent when modifying a design where some strands already have DNA sequences assigned, but in some cases it's not clear what to do. (e.g., what DNA sequence results when a length-5 strand with sequence AACGT is extended to be longer?)

DNA modifications

DNA modifications describe ways that various small molecules may be attached to synthetic DNA as part of the DNA synthesis process. Common DNA modifications include biotin (useful for binding to the protein streptavidin) and fluorophores such as Cy3 (useful for light microscopy). Modifications can be attached to the 5' end, the 3' end, or to an internal base.

A few pre-defined modifications are provided as examples in the Python scripting library. However, it is straightforward to implement a custom modification. For example, useful fields of a modification are `display_text`, which is displayed in the web interface (e.g., `B` for biotin; see Figure 1), and `idt_text`, the IDT code for the modification, used for exporting DNA sequences (e.g., `"/5Biosg/ACGT"`, which attaches a 5' biotin to the sequence `ACGT`).

Because it is common to attach one type of modification to several strands in a DNA design, modifications are defined at the top level of a DNA design, where they are given a string id, referenced on each strand that contains the modification.

2.2 scadnano file format

The following scadnano `.sc` file encodes the design in Figure 1 in a format called JSON, a commonly-used plain text format for describing structured data [9], with support in many programming language standard libraries. The format is not exhaustively described here, but the example shows how the JSON data maps to the data model described above.

```
{
  "grid": "square",
  "helices": [
    {"max_offset": 48, "grid_position": [0, 0]},
    {"max_offset": 48, "grid_position": [0, 1]}
  ],
  "modifications_in_design": {
    "/5Biosg/": {
      "display_text": "B",
      "idt_text": "/5Biosg/",
      "location": "5'"
    }
  },
  "strands": [
    {
      "color": "#0066cc",
      "sequence": "
          AACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACGTAACG",
      "domains": [
        {"helix": 1, "forward": false, "start": 8, "end": 24, "deletions": [20]},
        {"helix":0, "forward":true, "start":8, "end":40, "insertions":[[14,1],[26,2]]},
        {"loopout": 3},
        {"helix": 1, "forward": false, "start": 24, "end": 40}
      ],
      "is_scaffold": true
    },
    {
      "color": "#f74308",
      "sequence": "ACGTTACGTTACGTTTTACGTTACGTTACGTT",
      "domains": [
        {"helix": 1, "forward": true, "start": 8, "end": 24, "deletions": [20]},
        {"helix": 0, "forward": false, "start": 8, "end": 24, "insertions": [[14, 1]]}
      ]
    },
    {
      "color": "#57bb00",
      "sequence": "ACGTTACGTTACGTTACGCGTTACGTTACGTTAC",
      "domains": [
        {"helix": 0, "forward": false, "start": 24, "end": 40, "insertions":[[26,2]]},
        {"helix": 1, "forward": true, "start": 24, "end": 40}
      ],
      "5prime_modification": "/5Biosg/"
    }
  ]
}
```

2.3 Comparison to cadnano file format

The file format used by cadnano v2 is a grid of dimension (number of helices)$\times$(maximum offset) describing at each position whether a domain is present and the direction in which it is going. Additional information about *insertions* and *deletions* is given in a similar way.

An important goal of scadnano is to ensure interoperability with cadnano (see Section 3.9). Thus every cadnano design can be imported into scadnano. However, the converse is not true; scadnano's data model can describe features not present in cadnano.

1. cadnano does not have a way to encode loopouts, modifications, or gridless designs.
2. cadnano does not store DNA sequences in its file format.
3. cadnano has the constraint that helices with even index have the scaffold going forward and helices with odd index have the scaffold going backward. scadnano designs not following that convention cannot be encoded in cadnano.
4. cadnano does not explicitly encode the grid type, instead inferring it from the maximum helix offset: multiples of 21 represent the honeycomb grid, while multiples of 32 represent the square grid. To encode a scadnano design in cadnano's convention, each helix's maximum offset is modified to the lowest multiple of 21 or 32 fitting the design.

Converting a scadnano design to cadnano v2 is straightforward: lay out all domains of all strands in a (number of helices)×(modified maximum offset) grid. Maximum offsets have to be modified because of Item 4. However, converting a cadnano design to scadnano format is a bit more involved, requiring a connected components detection algorithm performed on the grid – similar to a depth-first search – in order to identify strands and their domains.

3 Features

3.1 Features shared with cadnano v2

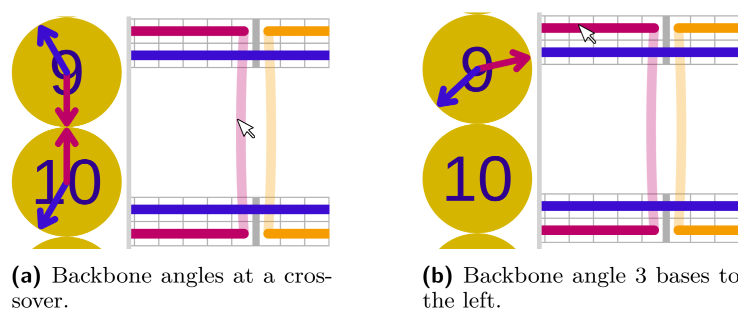
The web interface of scadnano is similar to cadnano (see Figure 1). Like cadnano, scadnano is optimal for structures consisting of parallel helices. On the left, the side view shows a cross-sectional view of the lattice where helices can be added to the design. The main view shows what the helix would look like going from left to right in the screen. Moving to the right in the main view is like moving "into the screen" in the side view.

DNA designs are drawn as they are often drawn in figures, with strands on a double-helix represented as straight lines that are connected to other helices by crossovers. Users can also add deletions and insertions (called *skips* and *loops* in cadnano) which means a strand has fewer or more bases than the interface's visually depicted length. Insertions and deletions help to use a regular spacing pattern – note the "major tick marks" every 8 bases on the helix – while allowing short regions to deviate and use more or fewer than the typical number of bases between two major tick marks. One feature scadnano adds to cadnano is the ability to customize the major tick marks, including non-regular spacing, e.g, alternating 10, 11, 10, 11 for single-stranded tiles [39, 42].

scadnano includes several "Edit modes", many similar to those of cadnano, shown in the top right corner of Figure 1. There are two main modes for editing, select mode and pencil mode, as well as several others explained in more detail in the scadnano documentation. Select mode allows users to select, resize, and delete items, just like in cadnano. (scadnano additionally allows users to copy and paste or move items; see Section 3.2). Pencil mode is used to create new objects such as helices, strands, or crossovers.

Users can assign DNA sequences to strands, and the complementary sequences for the bound strands are automatically computed. The common M13 DNA sequence is provided as a default for single-scaffold designs.

Although scadnano currently provides no 3D visualization, it does provide a primitive way to visualize the DNA backbone angles to help pick where to place crossovers; see Figure 2. This feature is slightly more flexible than the analogous feature in cadnano in that the user

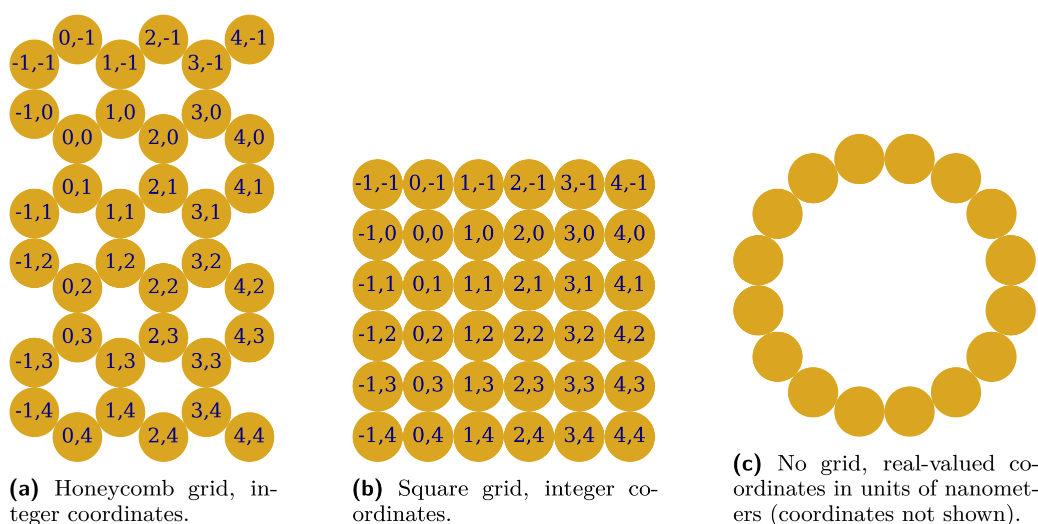


(a) Backbone angles at a crossover. **(b)** Backbone angle 3 bases to the left.

Figure 2 The side view displays the backbone angles to aid with crossover placement.

is allow to set the backbone angle at one base position to see what that implies about the backbone angle at other (typically nearby) base positions. For example, a user can "unstrain" the backbone at a crossover so that the backbone angles are perfectly aligned (see Figure 2a). The backbone angles at other positions are automatically computed (see Figure 2b).

The side and main view designs can be exported as SVG figures, and DNA sequences can be be exported into a CSV file, as well as formats recognized by the synthesis company IDT.



(a) Honeycomb grid, integer coordinates. **(b)** Square grid, integer coordinates. **(c)** No grid, real-valued coordinates in units of nanometers (coordinates not shown).

Figure 3 scadnano grids (hex grid not shown).

Like cadnano, helices can be placed in a square or honeycomb lattice, as shown in Figure 3a and Figure 3b. scadnano provides two more grids not available on cadnano: the hex grid (allowing helices in the "holes" of the honeycomb grid) and no grid; see Section 3.8.

The remainder of Section 3 describes features not shared with cadnano v2.

3.2 Copy and paste

A full DNA origami design using a standard 7249-base M13mp18 scaffold uses ≈ 200 staples, which are tedious to create manually. In scadnano, this process is accelerated by the

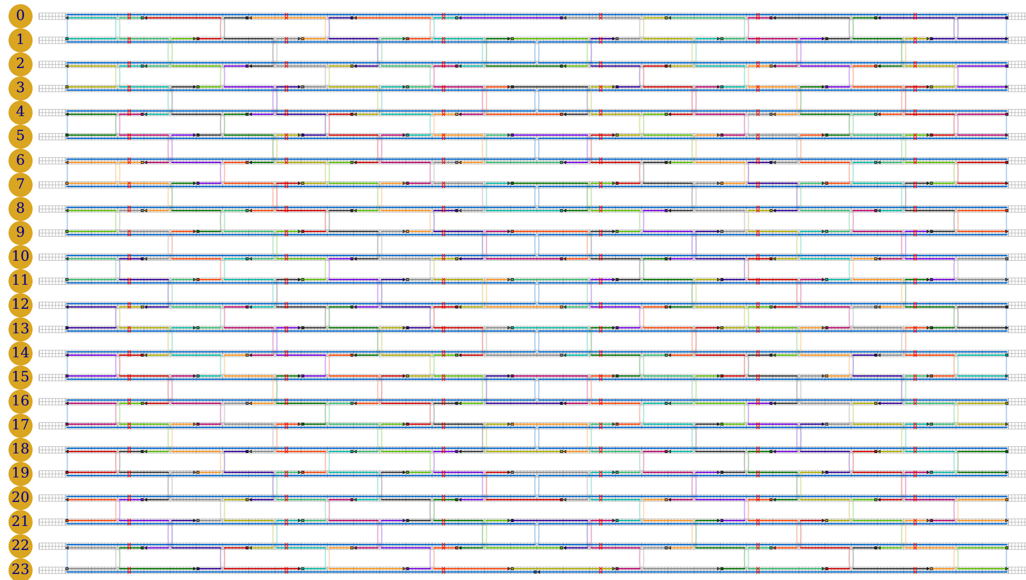


Figure 4 A standard 24 helix DNA origami rectangle design, with "twist-correction" [41].

copy/paste feature.[5] For instance, to create a vertical "column" of 24 staples in a 24-helix rectangle (see Figure 4), one would create 2 types of staples (plus some special cases near the top/bottom), copy/paste them to make 4, copy/paste those to make 8, then copy/paste the group of 8 two more times for a total of 24 staples. Since most of the design consists of horizontally translated copies of this column, it can be created quickly by copying and pasting the column.

3.3 Scripting library

The scadnano Python module allows one to write scripts for creating and editing scadnano designs. (Note that cadnano v2.5, unlike v2, does have a scripting library [2], though with incomplete documentation.) The module helps automate some of the tedious tasks involved in creating DNA designs, as well as making large-scale changes to them that are easier to describe programmatically than to do by hand in scadnano.

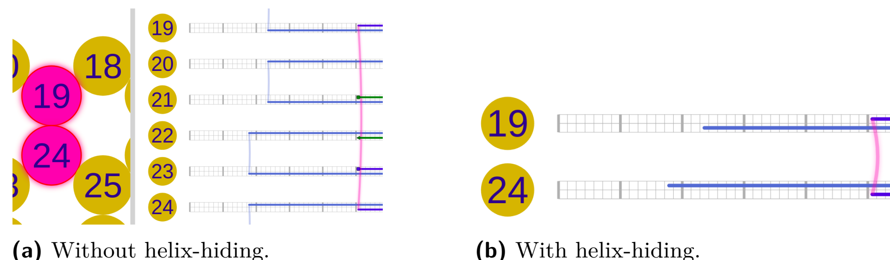
For example, the following is Python code generating the design in Figure 4, creating a `.sc` file with the design and a Microsoft Excel file with staple strand DNA sequences in a format ready to order from the DNA synthesis company IDT. It is perhaps unnecessary to read the code in detail; we provide it to demonstrate that "production-ready" designs can be created with relatively short and simple scripts. It follows the pattern described in the online tutorial (see first page).

[5] cadnano provides features to make large designs quickly, *autostaple* and *autobreak*, which are faster than copy/pasting strands, though they give less control over the outcome.

```
import scadnano as sc

def create_design():
    design = create_design_with_precursor_scaffolds()
    add_scaffold_nicks(design)
    add_scaffold_crossovers(design)
    scaffold = design.strands[0]
    scaffold.set_scaffold()
    add_precursor_staples(design)
    add_staple_nicks(design)
    add_staple_crossovers(design)
    add_twist_correcting_deletions(design)
    design.assign_m13_to_scaffold()
    return design

def create_design_with_precursor_scaffolds() -> sc.DNADesign:
    helices = [sc.Helix(max_offset=304) for _ in range(24)]
    scaffolds = [sc.Strand([sc.Domain(helix=helix, forward=helix%2 == 0, start=8, end
     =296)])
                 for helix in range(24)]
    return DNADesign(helices=helices, strands=scaffolds, grid=square)

def add_scaffold_nicks(design: sc.DNADesign):
    for helix in range(1, 24):
        design.add_nick(helix=helix, offset=152, forward=helix%2 == 0)

def add_scaffold_crossovers(design: sc.DNADesign):
    crossovers = []
    for helix in range(1, 23, 2): # scaffold interior
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=152, forward1=False))
    for helix in range(0, 23, 2): # scaffold edges
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=8, forward1=True, half=
     True))
        crossovers.append(
            sc.Crossover(helix1=helix, helix2=helix+1, offset1=295, forward1=True,half=
     True))
    design.add_crossovers(crossovers)

def add_precursor_staples(design: sc.DNADesign):
    staples = [sc.Strand([sc.Domain(helix=helix, forward=helix%2 == 1, start=8, end
     =296)])
               for helix in range(24)]
    for staple in staples:
        design.add_strand(staple)

def add_staple_nicks(design: sc.DNADesign):
    for helix in range(24):
        start_offset = 32 if helix % 2 == 0 else 48
        for offset in range(start_offset, 280, 32):
            design.add_nick(helix, offset, forward=helix%2 == 1)

def add_staple_crossovers(design: sc.DNADesign):
    for helix in range(23):
        start_offset = 24 if helix % 2 == 0 else 40
        for offset in range(start_offset, 296, 32):
            if offset != 152:  # skip crossover near seam
                design.add_full_crossover(helix1=helix, helix2=helix + 1,
                                          offset1=offset, forward1=helix % 2 == 1)

def add_twist_correcting_deletions(design: sc.DNADesign):
    for helix in range(24):
        for offset in range(27, 294, 48):
            design.add_deletion(helix, offset)

def export_idt_plate_file(design: sc.DNADesign):
    for strand in design.strands:
        if strand != design.scaffold:
            strand.set_default_idt(use_default_idt=True)
    design.write_idt_plate_excel_file(use_default_plates=True)

if __name__ == "__main__":
    design = create_design()
    export_idt_plate_file(design)
    design.write_scadnano_file()
```

3.4 Hiding helices to aid 3D design

The 2D main view in scadnano distorts the relative positions of the helices if they do not form a flat 2D shape as in Figure 4. For example, consider Figure 5. Helices 19 and 24, though adjacent (see side view), appear far apart in the main view. Thus crossovers between these helices, while appearing to stretch over a long distance (Figure 5a), are the same length as any other crossover (just a single phosphate group between two DNA bases).



(a) Without helix-hiding. **(b)** With helix-hiding.

Figure 5 Two helices in a design, 19 and 24, are adjacent in the side view (i.e., in the actual 3D structure) but not in the main view. The selected crossover appears "long-range" in Figure 5a, but "short-range" in Figure 5b.

This can make it difficult to analyze and edit 3D designs. For example, consider the squarenut design from the original 3D origami paper [23] (see Figure 6a). This design is difficult to visualize because the 2D view is not representative of the 3D positions of the actual DNA helices, in no small part because of the "cobweb" of crossovers that results.

To aid in visualization, scadnano can display only selected helices (see Figure 6b). Helix 19 and 24 in Figure 5b can be seen in the side view are actually adjacent in 3D space. When other helices are hidden, helices 19 and 24 are displayed adjacently in the main view.

cadnano puts all helices immediately adjacent to each other in the order they are displayed in the main view. scadnano uses the distance between helices (as determined by their grid position or gridless 3D position) to determine distances. Helices are displayed in order of their index field `idx` (unless `helices_view_order` is specified to alter this order), but two helices adjacent in this order will have a vertical distance between them in the main view proportional to the distance as determined by the grid position or gridless 3D position.

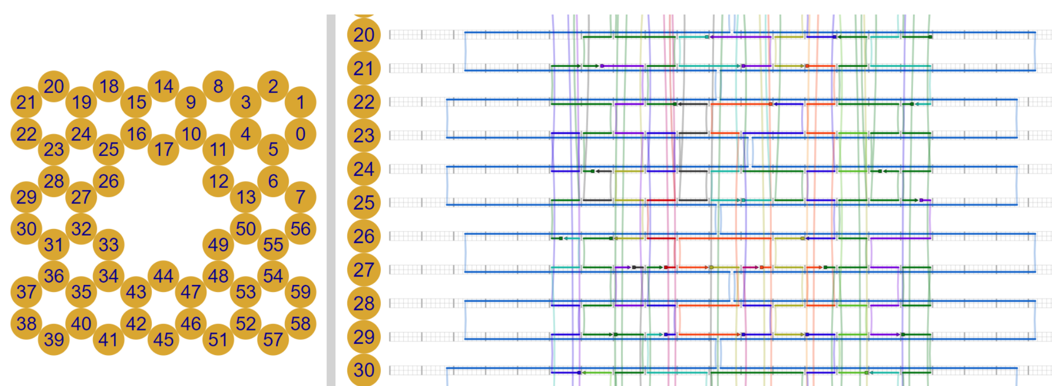
3.5 Single-stranded loopouts

scadnano allows a type of single-stranded domain not associated to any helix, called a *loopout*, used to describe common single-stranded features such as hairpins. In cadnano users would need to make a "fake" helix if they want to add a single-stranded DNA. For some designs, this creates awkward artifacts such as long-range crossovers to reach the fake helix.

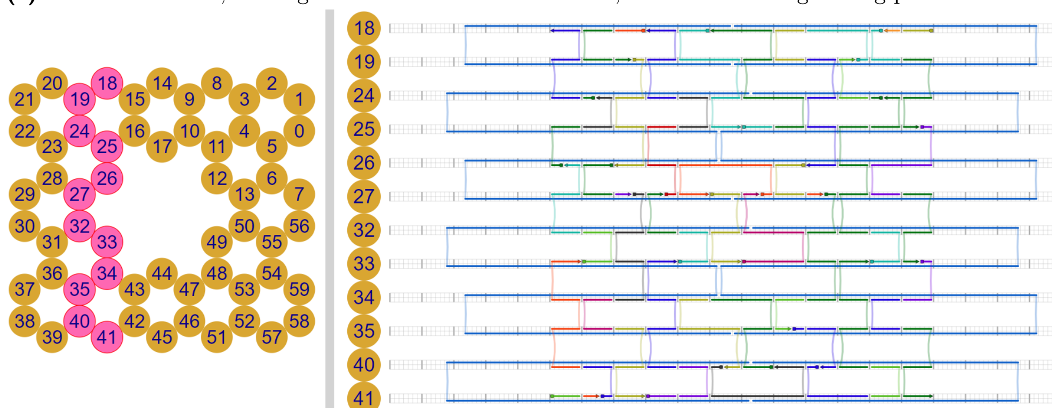
3.6 DNA modifications

scadnano supports for DNA modifications, such as biotin or Cy3 [8]. Figure 7a shows an example of biotin modifications to the 5' end of some staples in a 16-helix DNA origami. Users can specify a string such as `"O"` to represent the modification in the web interface.

The aspect ratio is proper for 2D origami with helices all stacked in the square lattice, helping to place modifications and visualize their relative positions to scale. Compare the scadnano display in Figure 7a to the AFM image in Figure 7b. Currently, only a few pre-loaded modifications are provided, but users can describe custom modifications.



(a) All helices shown, causing the dreaded *crossover cobweb*, like laser beams guarding priceless art.



(b) Restricted subset of helices displayed: only relevant helices and crossovers are shown.

Figure 6 Squarenut 3D origami [23], a typical 3D origami difficult to visualize in a 2D projection.

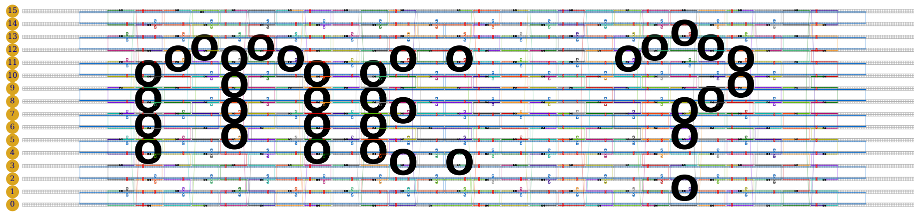
3.7 Unused fields

In order to maximize interoperability with other tools, scadnano allows arbitrary fields to be included in a scadnano `.sc` file. Any fields that it does not recognize are simply ignored. However, they are stored and written back out when the file is saved. Thus, "light" editing of scadnano files is possible that will preserve fields used by other programs. For example, codenano [5] allows an optional field `label` on each strand, which will be preserved for each strand by scadnano while editing other aspects of the design.

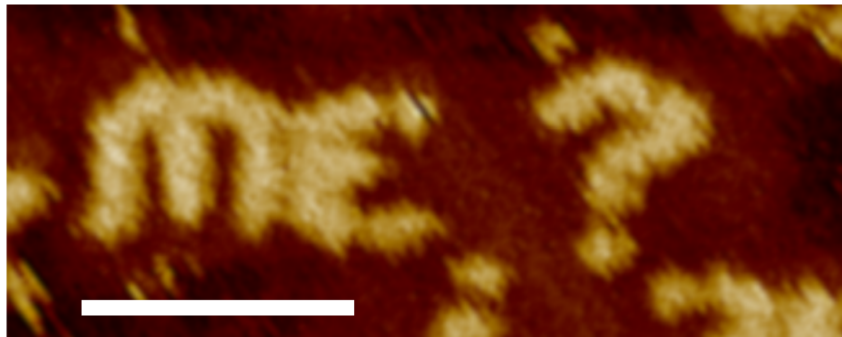
3.8 Gridless helix placement

scadnano includes the option to use no grid; see Figure 3c. This allows more flexible helix placement, where helix centers can be placed at any real-valued (i.e., floating-point) (z, y) coordinate. This feature is useful for some designs that do not align nicely with the standard square or honeycomb lattice. In the absence of a grid, coordinates of helices are specified in nanometers. By default, the distance between each DNA helix center is 3 nm.[6]

[6] The accepted measurement of the DNA double-helix diameter is ≈ 2 nm. However, AFM images show that in 2D square-lattice DNA origami designs, an origami with n helices will have height in nanometers of approximately $3 \cdot n$ due to electrostatic repulsion between neighboring helices.



(a) biotin DNA modifications on the 5' end of some staples, displayed in scadnano.



(b) The same design imaged with atomic force microscopy (AFM), with streptavidin added to visualize the biotin locations. (scale bar: 50 nm) (image source: `https://web.cs.ucdavis.edu/~doty/papers/#proposal`)

Figure 7 An example of a design containing biotin modifications.

3.9 Interoperability with cadnano

Interoperability with cadnano (version 2) is an important goal of the project. Both the scadnano GUI and Python module provide functionality that allows users to import/export a design from/to cadnano. All cadnano (version 2) designs can be imported in scadnano. However, because of fundamental differences between the way cadnano and scadnano encode designs, some scadnano designs cannot be converted cadnano (see Section 2.3).[7]

4 Software architecture

4.1 Two codebases

The codebase for scadnano is split into two pieces: the Python scripting library, and the web interface. Unfortunately, some algorithmic functionality is duplicated between them. We chose Python as the scripting language because it is easy to learn and already familiar to many physical scientists likely to use scadnano. However (despite innovations such as Pyodide [11], Skulpt [15], and Brython [1]), Python is not well-suited for *front-end* web programming, where the code is executed in the browser rather than on a server. A design goal of scadnano is to do as much work as possible in the browser.

The web interface is instead implemented using the Dart programming language [6], a modern, strongly-typed, object-oriented language that can be compiled to Javascript, the *lingua franca* of web browsers. In order to make the Python scripting library as easy to use as possible (no dependence on Dart libraries) and to keep the web interface as fast as possible

[7] These constraints are described in the documentation: `https://scadnano-python-package.readthedocs.io/en/latest/index.html#interoperability-cadnano-v2`

and avoid the need to farm out computation to a server, some algorithms (e.g., computing complementary DNA sequences of strands when they are bound to another strand that has had a DNA sequence assigned to it) are implemented in both libraries.

However, we intend for the file format to be decoupled from the scripting and web-based programs that manipulate it. Indeed, another tool called codenano [5] uses essentially the same file format as scadnano, although that program is written in Rust and has the user specify the design by writing Rust code.

4.2 Unidirectional data flow in graphical user interface code

Graphical user interface software, inherently asynchronous and non-sequential, is notoriously difficult to reason about. Whole classes of bugs exist that do not plague programs with only sequential logic. The open-source software community has developed many tools to aid in such design. The *model-view-controller* (MVC) architecture is almost as old as graphical interfaces themselves, dating to the 1970s [29]. However, MVC is not very well-defined, particularly the controller part, and still lends itself to common bugs.

A more recent innovation, originating within the past decade, goes under a few names, such as *model-view-update*, *the Elm architecture* [7], or *unidirectional data flow* [16]. Several variants exist implementing the idea. We chose a popular pair of technologies, React [12] and Redux [14]. They are designed for Javascript, but since Dart compiles to Javascript, they can be used with Dart with appropriate wrapping libraries [10, 13].

The cited links go into detail about the architecture; we summarize it briefly here for the curious. Briefly, all application *state* is stored in a single immutable object. (In scadnano, this includes the entire DNA design, as well as more ephemeral UI state, such as which strands are currently selected.) Immutability is a powerful concept in programming, allowing one to share an object between many concurrent processes without worrying that one process will modify it in ways unexpected by the other processes. The global state object is a tree (cycles are difficult to handle with immutable objects). The *view* (what the user sees on the screen) is specified as a deterministic function of the state. This greatly reduces the "surface area" where bugs can (and reliably do) occur: the application does not have to contain code stating how to *modify* the view in response to any possible change in the state. It merely says what the *entire* view should be, as a function of the *entire* state.

Changes to the application state are expressed using the Command pattern [25] by dispatching an *action* describing that the state should change. The application responds to the action by computing the new state as a deterministic function of the old state and the action. The view redraws itself, but optimizations ensure only the parts that depend on changed state will actually be redrawn.

This decoupling of actions that change state (and the sometimes complex logic behind them), and views that draw themselves as a function of a single state, is the key to making it straightforward to implement new features without introducing bugs. It's not foolproof; bugs do occur. There is also a nontrivial computational cost: the React library compares the old state to the new to determine which subtrees actually changed (determining which parts of the view actually need to re-render), a potentially expensive operation.

However, we find it is worth the computational cost for the benefit of reliability. We believe it will make it easier to maintain scadnano, fix bugs, and add features in the future.

Both the Python package and the Dart web interface are open-source software to which anyone can contribute. Both repositories have a CONTRIBUTING document explaining how to contribute to the projects, following the git model of making a separate branch, adding the change, and doing a pull request to merge the changes. Both repositories are currently maintained by the first author, who reviews all pull requests.

5 Conclusion

The goal of scadnano is to reproduce the usefulness of cadnano for designing large-scale DNA structures in a web app with a well-documented, easy-to-use scripting library. It is ready to use for designing DNA structures, although some work remains to bring it up to a more polished state. The issues page of each repository (see first page) shows many bugs and feature enhancements that have not yet been addressed.

scadnano excels where cadnano excels: in describing DNA structures where all DNA helices are in parallel. A broader range of DNA nanostructures exists, such as wireframe designs [19, 44] and curved DNA origami shapes [22, 27]. A 2D projected view can describe these, but more awkwardly than a 3D view. Since the chief goal of scadnano is to remain easy to use and responsive to bug reports and feature requests within the current scope of scadnano, it will remain for the near-term future as a tool primarily for designs that are straightforward to visualize in 2D. We outline possible future work:

export to other file formats. Currently, scadnano can export to the cadnano v2 file format, and it can export DNA sequences in either a comma-separated value (CSV) file, which can be processed by the user's custom scripts, or in a few formats recognized by the DNA synthesis company IDT (Integrated DNA Technologies, Coralville, IA, `https://www.idtdna.com`). It should be straightforward to export to formats recognized by other DNA synthesis companies (e.g., Bioneer), or other DNA nanotech software (e.g., oxDNA).

helices rotated in the main view plane. Some 2D structures do not have all helices in parallel, for example DNA origami implementations of 4-sided tiles [37], or flat origami "stiffened" by a second layer of perpendicular helices [36]. We are exploring design ideas for supporting this in a way "natural" for editing in the 2D view. In particular, copy/paste and moving of strands spanning multiple helices makes most sense for groups of helices that are parallel. One idea is to let a design specify several helix *groups*, where all helices within a group are parallel, but the groups have different rotations and translations. (For example, there would be two groups for [36] and two or four groups for [37].)

3D visualization. cadnano has never been ideal for visualizing arbitrary 3D structures, and neither is scadnano currently. It may remain the case that the ideal way to visualize 3D structures is to export the design to another tool specialized for the job, such as codenano [5], CanDo [4], or oxDNA [35]. However, WebGL provides a powerful platform for visualizing 3D structures, used by other software such as oxDNA and codenano. In fact, since codenano is itself implemented as a web app (written in Rust that is compiled to WebAssembly, which is itself callable from Javascript), it should be possible to implement the 3D visualization features of codenano as a library that scadnano can call.

DNA design database. Communication of DNA designs through the Supplementary Information of a journal remains an ad hoc method. A centralized database of DNA designs would benefit the community. We hope that the scadnano/codenano file format is sufficiently expressive to describe any such design. However, such a database need not have anything to do with the scadnano website itself.

collaborative editing. Collaborative editing tools such as Google Docs make use of a recently developed technique known as a *conflict-free replicated data type* (CRDT) [34]. It is conceivable that a CRDT representation of a DNA design could enable remote collaborators to simultaneously view and edit a DNA design.

References

1. Brython. https://brython.info/.
2. cadnano v2.5. https://github.com/cadnano/cadnano2.5.
3. cadnano v2.5 Python API. https://cadnano.readthedocs.io/en/master/scripting.html.
4. Cando. https://cando-dna-origami.org/.
5. codenano. https://dna.hamilton.ie/2019-07-18-codenano.html.
6. Dart programming language. https://dart.dev/.
7. Elm programming language. https://elm-lang.org/.
8. IDT DNA modifications. https://www.idtdna.com/pages/products/custom-dna-rna/oligo-modifications.
9. Json (javascript object notation). https://www.json.org/json-en.html.
10. Overreact Dart library. https://pub.dev/packages/over_react.
11. Pyodide. https://github.com/iodide-project/pyodide.
12. React Javascript library. https://reactjs.org/.
13. Redux Dart library. https://pub.dev/packages/redux.
14. Redux Javascript library. https://redux.js.org/.
15. Skulpt. https://skulpt.org/.
16. Unidirectional data flow in Redux. https://redux.js.org/basics/data-flow.
17. SAMSON, the open molecular modeling platform. https://www.samson-connect.net, 2019.
18. Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, July 2015. doi:10.1038/nature14586.
19. Erik Benson, Abdulmelik Mohammed, Johan Gardell, Sergej Masich, Eugen Czeizler, Pekka Orponen, and Björn Högberg. DNA rendering of polyhedral meshes at the nanoscale. *Nature*, 523(7561):441–444, 2015.
20. Gourab Chatterjee, Neil Dalchau, Richard A Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature nanotechnology*, 12(9):920, 2017.
21. Elisa de Llano, Haichao Miao, Yasaman Ahmadi, Amanda J. Wilson, Morgan Beeby, Ivan Viola, and Ivan Barisic. Adenita: Interactive 3D modeling and visualization of DNA nanostructures. Technical report, bioRxiv, 2019. doi:10.1101/849976.
22. Hendrik Dietz, Shawn M Douglas, and William M Shih. Folding DNA into twisted and curved nanoscale shapes. *Science*, 325(5941):725–730, 2009.
23. Shawn M Douglas, Hendrik Dietz, Tim Liedl, Björn Högberg, Franziska Graf, and William M Shih. Self-assembly of DNA into nanoscale three-dimensional shapes. *Nature*, 459(7245):414–418, 2009.
24. Shawn M Douglas, Adam H Marblestone, Surat Teerapittayanon, Alejandro Vazquez, George M Church, and William M Shih. Rapid prototyping of 3D DNA-origami shapes with caDNAno. *Nucleic Acids Research*, 37(15):5001–5006, 2009. URL: https://cadnano.org/.
25. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.
26. Hongzhou Gu, Jie Chao, Shou-Jun Xiao, and Nadrian C Seeman. A proximity-based programmable DNA nanoscale assembly line. *Nature*, 465(7295):202–205, 2010.
27. Dongran Han, Suchetan Pal, Jeanette Nangreave, Zhengtao Deng, Yan Liu, and Hao Yan. DNA origami with complex curvatures in three-dimensional space. *Science*, 332(6027):342–346, 2011.
28. Hyungmin Jun, Xiao Wang, William Bricker, Steve Jackson, and Mark Bathe. Rapid prototyping of wireframe scaffolded DNA origami using ATHENA. Technical report, bioRxiv, 2020. doi:10.1101/2020.02.09.940320.
29. Glenn Krasner and Stephen Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of object-oriented programming*, 1, 1988.

30 Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1), November 2011. doi:10.1186/1748-7188-6-26.

31 Christopher Maffeo and Aleksei Aksimentiev. MrDNA: A multi-resolution model for predicting the structure and dynamics of nanoscale dna objects. *bioRxiv*, 2019. doi:10.1101/865733.

32 Dionis Minev, Christopher M. Wintersinger, Anastasia Ershova, and William M Shih. Robust nucleation control via crisscross polymerization of DNA slats. Technical report, biorXiv, 2019. URL: https://www.biorxiv.org/content/10.1101/2019.12.11.873349v1.

33 Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

34 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011: Symposium on self-stabilizing systems*, pages 386–400, 2011.

35 Benedict EK Snodin, Ferdinando Randisi, Majid Mosayebi, Petr Šulc, John S Schreck, Flavio Romano, Thomas E Ouldridge, Roman Tsukanov, Eyal Nir, Ard A Louis, and Jonathan P. K. Doye. Introducing improved structural properties and salt dependence into a coarse-grained model of DNA. *The Journal of chemical physics*, 142(23):234901, 2015.

36 Anupama J Thubagere, Wei Li, Robert F Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, Erik Winfree, and Lulu Qian. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

37 Grigory Tikhomirov, Philip Petersen, and Lulu Qian. Programmable disorder in random DNA tilings. *Nature nanotechnology*, 12(3):251, 2017.

38 Petr Šulc, Flavio Romano, Thomas E. Ouldridge, Lorenzo Rovigatti, Jonathan P. K. Doye, and Ard A. Louis. Sequence-dependent thermodynamics of a coarse-grained DNA model. *The Journal of Chemical Physics*, 137(13):135101, 2012. doi:10.1063/1.4754132.

39 Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.

40 Erik Winfree, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, 1998.

41 Sungwook Woo and Paul WK Rothemund. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature chemistry*, 3(8):620, 2011.

42 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567:366–372, 2019. doi:10.1038/s41586-019-1014-9.

43 Joseph N. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. Nupack: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32(1):170–173, 2011. doi:10.1002/jcc.21596.

44 Fei Zhang, Shuoxing Jiang, Siyu Wu, Yulin Li, Chengde Mao, Yan Liu, and Hao Yan. Complex wireframe DNA origami nanostructures with multi-arm junction vertices. *Nature nanotechnology*, 10(9):779, 2015.