nature portfolio

https://doi.org/10.1038/s41596-022-00688-5

Supplementary information

Design and simulation of DNA, RNA and hybrid protein–nucleic acid nanostructures with oxView

In the format provided by the authors and unedited

Supplemental Information for "Design and simulation of DNA, RNA, and hybrid protein-nucleic acid nanostructures with OxView"

Joakim Bohlin,[1] Michael Matthies,[2] Erik Poppleton,[2] Jonah Procyk,[2] Aatmik Mallya,[2] Hao Yan,[2] and Petr Šulc[2,*]

[1] *Clarendon Laboratory, Department of Physics, University of Oxford, Parks Road, Oxford OX1 3PU, UK.*
[2] *Center for Molecular Design and Biomimetics at the Biodesign Institute and School of Molecular Sciences, Arizona State University, Tempe, Arizona 85287, United States*

SI. OXVIEW FILE FORMAT

Oxview has a JSON-based file format to save and load designs. An .oxview file is structured in a simple hierarchy, with a main object containing a "systems" list together with possible metadata such as simulation box size.

An online description of the file format is available at `https://github.com/sulcgroup/oxdna-viewer/blob/master/file-format.md`

```
{
  "box": [100, 100, 100],
  "systems": []
}
```

Listing S1. The outer layer of the oxView JSON hierarchy

Positions are noted in oxDNA units, where 1 distance unit equals 0.8518 nanometers.

System

Each system needs to contain a unique system index "id" and a list of strands.

```
{
  "id": 0,
  "strands": []
}
```

Listing S2. The system layer of the oxView JSON hierarchy

Strand

The attributes of each strand should be: "id" - a unique strand index, "end3" - the monomer index of the 3' end of the strand, "end5" - the monomer index of the 5' end of the strand, "class" - the strand type (currently "NucleicAcidStrand" and "Peptide" are supported), and "monomers" - a list of monomers.

Circular strands should still have "end3" and "end5" specified, as this will indicate where to start traversal. Just make sure that the monomer with id "end3" has an "n3" of "end5" and the "end5" monomer has an "n5" of "end3".

```
{
  "id": 0,
  "end3": 0,
  "end5": 63,
  "class": "NucleicAcidStrand",
  "monomers": []
}
```

Listing S3. The strand layer of the oxView JSON hierarchy

* psulc@asu.edu

Monomer

The attributes of each monomer should be: “id” - a unique monomer index, “type” - the monomer type (e.g. A, T, C, or G if DNA), “class” - the monomer type (currently “DNA”, “RNA” and “AA” (amino acid) are supported), “p”: center of mass position (in oxDNA coordinates), “a1”: backbone vector, and “a3”: stacking vector.

Monomers can also include “n5” - the monomer index of the 5’ neighbor, “n3” - the monomer index of the 3’ neighbor, “bp” - the monomer index of the paired nucleotide (for DNA and RNA), “cluster” - index of cluster group that the monomer belongs to, and “color” a base 10 representation of a hexadecimal color (used to set custom coloring).

```
{
  "id": 0, "type": "C", "class": "DNA",
  "p": [0,0,0],
  "a1": [0.8271594,0.5538015,0.0954520],
  "a3": [0,0,1],
  "n3": 63, "n5": 1, "bp": 95,
  "cluster": 1, "color": 3633362
}
```

Listing S4. The monomer layer of the oxView JSON hierarchy

Complete example

Listing S5 shows an example of a complete oxview file, describing two base pairs of a DNA helix.

```
{
  "date": "2021-08-23T08:38:04.553Z",
  "box": [10, 10, 10],
  "systems": [{
      "id": 0,
      "strands": [
        {
          "id": 0,
          "monomers": [
            {
              "id": 2,
              "type": "A",
              "class": "DNA",
              "p": [-0.3518234193325043, -0.48602294921875, -0.19488525390625],
              "a1": [0.586372371762991, 0.810089111328125, 0],
              "a3": [0, 0, -1],
              "n3": 0,
              "cluster": 1,
              "color": 16777215,
              "bp": 3
            },
            {
              "id": 0,
              "type": "A",
              "class": "DNA",
              "p": [0, -0.5999755859375, 0.19488525390625],
              "a1": [0, 1, 0],
              "a3": [0, 0, -1],
              "n5": 2,
              "cluster": 2,
              "bp": 1
            }
          ],
          "end3": 0,
          "end5": 2,
          "class": "NucleicAcidStrand"
        },
        {
          "id": 1,
          "monomers": [
            {
              "id": 1,
```

```
            "type": "T",
            "class": "DNA",
            "p": [0, 0.5999755859375, 0.19488525390625],
            "a1": [0, -1, 0],
            "a3": [0, 0, 1],
            "n3": 3,
            "cluster": 2,
            "color": 16777215,
            "bp": 0
          },
          {
            "id": 3,
            "type": "T",
            "class": "DNA",
            "p": [0.3518234193325043, 0.48602294921875, -0.19488525390625],
            "a1": [-0.586372371762991, -0.810089111328125, 0],
            "a3": [0, 0, 1],
            "n5": 1,
            "cluster": 1,
            "color": 16711680,
            "bp": 2
          }
        ],
        "end3": 3,
        "end5": 1,
        "class": "NucleicAcidStrand"
      }
    ]
  }
 ]
}
```


Listing S5. Example of complete oxview file

SII. OXDNA FILE FORMAT

A structure in oxDNA is defined by two plaintext files. When loading these files into oxView, the trajectory and topology must be selected and opened together (whether via drag-and-drop or via the open dialogue). The first is the 'topology file' (file extension .top). The topology will have a one-line header followed by one line per particle. The header has either two (for DNA or RNA alone) or four (for simulations including an ANM) space-separated values. These are the number of nucleotides and the number of strands or the total number of particles, the total number of strands, the number of nucleotides, the number of amino acids, and the number of DNA/RNA strands, respectively. On each subsequent line is a space-separated list of particle parameter: the strand ID (positive integers for DNA/RNA, negative integers for proteins), the particle identity, which can either be the standard 1-letter code or an integer (integers which sum to 3 are allowed to base pair in the DNA/RNA models). The next two are the particle ids of the 3' and 5' neighbors or -1 in the case of a strand end (note that this is inverse from the standard DNA/RNA convention) or the N and C neighbors (which is the standard protein convention). For amino acids, there are then the ids of any number of other amino acid acids which they are bonded to in the ANM model.

for example, the first three lines of the linact.top file generated in Procedure 1 looks like:

```
23420 362
1 T -1 1
1 G 0 2
```

Listing S6. The first three lines of an oxDNA topology file showing the header a strand end and a nuclotide inside a strand.

The second file is the 'configuration file' (file extension .dat, .conf or .oxdna). The configuration has a 3-line header containing the current timestep of the simulation, the simulation box dimensions, and the potential, kinetic and total energy of the simulation at the current step. The header is then followed by one line per nucleotide where each line has 15 space-separated values containing the center of mass position, the hydrogen-bonding face orientation, the stacking face orientation, the translational velocity and the rotational velocity. A 'trajectory file' is simply many configurations appended together, each one starting again with the three-line header.

For example, the first five lines of the linact.dat file generated in Procedure 1 with the precision reduced to 5 decimals looks like:

```
t = 0
b = 479 479 479
E = 0 0 0
6.42161 6.33274 20.87749 0.58441 -0.81145 0 0 0 -1 0 0 0 0 0 0
6.75381 6.44559 20.48773 0.03074 -0.99952 0 0 0 -1 0 0 0 0 0 0
```

Listing S7. The first five lines of an oxDNA configuration file showing the header and two nucleotide positions/orientations.

SIII. SCRIPTING FOR NANOSTRUCTURE DESIGN

The following text is intended as a cookbook, demonstrating common examples of the oxView scripring API. Please refer to the source code and manual page on https://github.com/sulcgroup/oxdna-viewer for more details. Basic knowledge of the JavaScript programming language is helpful.

As described in the main text, in addition to the options available in the graphical interface, oxView is scriptable using the browser developer console. To enter a JavaScript program that edits the structure, follow this procedure:

1. Load a structure that you want to edit into OxView
2. Open a JavaScript console. On Chrome it is accessible through pressing **CTRL + SHIFT + J** on Windows or Linux, and by pressing **Command+ Option + J** on Mac OS. On Firefox, press **CTRL + SHIFT + I**.
3. Paste the script into the opened console and press enter.

OxView's object hierarchy follows the structure of an oxDNA simulation file. Each loaded file is assigned to be a new **System** object. Each **System** is composed of **Strand** objects stored in the **strands** property. These are abstractions of nucleic acid strands (**NucleicAcidStrand**) and peptides(**Peptide**), which are containers for **Nucleotide** and **AminoAcid** elements which will be called **BasicElement** from here forth and are returned as an Array uppon the **getMonomers()** strand method. There are three ways one can interact with a loaded object in oxView using the developers console to get to the **BasicElement**'s:

1. **selectedBases** - A JS set object, which contains the currently selected elements.
2. **systems** - A list of currently loaded systems.
3. **api.getElements** - A function which returns elements with the provided ids.

A. selectedBases

All **BasicElement** objects which are currently selected on the scene are stored in the **selectedBases** Set. To access individual elements it is easiest to convert the Set object to an array:

```
let selected = Array.from(selectedBases);
selected[0] // first selected element
```

Listing S8. Access the first selected element.

B. systems - variable

Each instance of oxView contains a list of **System** objects. These can be accessed through the **systems** variable. As described before, the **getMonomers()** method returns an Array of the **BasicElement**'s comprising a given strand.

```
let monomers = systems[0].strands[0].getMonomers();
monomers[0]; // the first BasicElement of the first strand.
```

Listing S9. Accessing the first BasicElement of the first strand of the first system.

C. api.getElements

The most common way to refer to elements when setting up oxDNA simulations is by their absolute index in the oxDNA topology file. oxView provides this functionality by using the **api.getElements** - Function call.

```
let first = api.getElements([0]);
```

Listing S10. Accessing the first BasicElement as referenced in the topology file.

D. Examples

Building on this knowledge let us walk through a couple of practical examples.

1. *Coloring a list of particles*

A typical problem when visualizing nucleic acid nanostructures is identifying problematic strands. If a staple strand is too short it may not bind stably to the structure and a too long strand is hard to synthesize. The following code entered in the developer console creates a color scheme where the strands with a length below 18 nucleotides are colored red, 18-50 nucleotides are considered optimal staple length and are colored green, 50-500 nucleotides are considered to long staples and colored orange. Strands above 500 nucleotides are considered as scaffold stands and colored blue.

```
// first we define the different strand length categories
let minStapleLength   = 18;
let maxStapleLength   = 50;
let minScaffoldLength = 500;
// a color scheme for the respective strands
let colors = {
  "small"   : new THREE.Color('red'),
  "optimal" : new THREE.Color('green'),
  "long"    : new THREE.Color('orange'),
  "scaffold": new THREE.Color('blue')
};
// we assume we have only 1 design loaded
systems[0].strands.forEach(strand=> {
    //  get the monomers of the current strand
    let monomers = strand.getMonomers();
    // and the strand length
    let length = strand.getLength();

    //  Assign the strand to a key in the color scheme
    let selector = "scaffold";
    if(length < minStapleLength)
      selector = "small";
    if(length >= minStapleLength && length <= maxStapleLength)
      selector = "optimal";
    if(length > maxStapleLength && length <= minScaffoldLength)
      selector = "long";
    // color the strand
    colorElements(colors[selector], monomers);
});
render();
```

Listing S11. Coloring strands by length (JS code)

2. *Printing the sequence of selected stands*

Building on the previous example, one might be interested in obtaining the sequence of selected strands for a given design. The result of this script is sequentially printed onto the developer console.

```
let strandSet = new Set(); // Set objects store only 1 instance of every element provided
```

```
selectedBases.forEach(nucleotide => strandSet.add(nucleotide.strand)); // figure out the strands
    iterating through all selectedBases
strandSet.forEach(strand => console.log(strand.getSequence())); // print the strand sequences
```

Listing S12. Get sequence of selected strands (JS code)

3. Visualizing the center of mass for a provided trajectory

Another common problem during trajectory visualisation is to highlight some structural properties. oxView provides a special API, which allows to bind the computation/visualisation of sertain parameters to one of the oxView update functions. Examples of update functions are **render**, **trajReader.nextConfig** or **trajReader.previousConfig**. Any function can be bound using the **api.observable.wrap** call as shown in Listing 6. To get one started oxView provides a couple of useful example classes. The **api.observable.CMS** class provides an easy way to visualize the center of mass of provided set of particles. And the **api.observable.Track** creates a trail following the particle provided position. The following code visualizes the track of the center of mass of a selected system.

```
let cms = new api.observable.CMS(selectedBases, 1, 0xFF0000);
let track =  new api.observable.Track(cms);
const updateFunc = ()=>{
    cms.calculate();
    track.calculate();
};
render = api.observable.wrap(render, updateFunc);
render();
```

Listing S13. Creating an order parameter (JS code)

4. Constructing a crystal cluster from a provided origami design

Here we provide the full JavaScript source code for the example in Figure 4 in the main text. The code below creates a cubic lattice composed of 3D DNA wireframe structures. It copies and pastes the individual origamis to create the lattice geometry and then connects the neighboring origamis via their single-stranded overhangs.

```
// adjust box
box.set(500,500,500);
// function translating a base index to a connection call
function connect(f, t){
    //receive first strand
    let s1 = elements.get(f).strand;
    //receive second strand
    let s2 = elements.get(t).strand;
    //connect the 2 strands by a duplex patch
    edit.interconnectDuplex3p(s1,s2);
}
//grid id to index storage
let d = {};
// we assume that the first system is the one we want to copy around
systems[0].select();
// we store the number of bases comprising the system to calculate the offset of the patch
    positions
const n_elements = selectedBases.size;
// compute the center of mass of the octahedron origami
let cms = new THREE.Vector3(0,0,0);
selectedBases.forEach( base =>{
    cms.add(base.getPos());
});
cms.divideScalar(selectedBases.size);

// prepare to copy around
cutWrapper();
// index of the currently generated origami
let idx = 0;
// we construct a grid of 3x3x3
for(let k=0;k < 3; k++){
```

```
    for(let j = 0; j < 3; j++){
        for(let i=0; i < 3; i++){
            //build up the index of the origami in the grid to use for offset
            d[`${i},${j},${k}`]= idx++;
            //print progress
            console.log(i,j,k)
            //paste in a new structure, true keeps the position
            pasteWrapper(true);
            //make sure everything is its own cluster
            selectionToCluster();
            //copy our computed cms value
            let cms_c = new THREE.Vector3().copy(cms);
            //compute the offset position
            cms_c.set(cms.x +80*i,cms.y+80*j,cms.z+80*k);
            //move the origami
            translateElements(selectedBases, cms_c);
        }
    }
}
// lets build up the connections in the grid
for(let k=0;k < 3; k++){
    for(let j = 0; j < 3; j++){
        for(let i=0; i < 3; i++){
            // retrieve the index of the origami we are connecting
            const self_idx = d[`${i},${j},${k}`];
            // an origami in a cubic lattice has 3 neighbors
            // so we get their indexes
            const right    = d[`${i+1},${j},${k}`];
            const top      = d[`${i},${j+1},${k}`];
            const north    = d[`${i},${j},${k+1}`];
            // print progress
            console.log(self_idx, right,top, north);
            // compute offset for the origami we are working on
            const s = self_idx*n_elements;
            if(right){ // if we have a right neighbor
                // compute the offset for it
                const r = right*n_elements;
                // connect all 4 patches to the current origami in the grid
                // indices are derived from the initial origami design, we want to copy around
                connect(s+12420, r+8414);
                connect(s+9994 , r+11794);
                connect(s+7318 , r+11168);
                connect(s+7254 , r+9446);
            }
            if(top){ // if we have a top neighbor
                // compute the offset for it
                const t = top*n_elements;
                // connect all 4 patches to the current origami in the grid
                // indices are derived from the initial origami design, we want to copy around
                connect(s+7866 , t+6692);
                connect(s+11858, t+6144);
                connect(s+11232, t+10058);
                connect(s+12968, t+10606);
            }
            if(north){ // if we have a north neighbor
                // compute the offset for it
                const n = north*n_elements;
                // connect all 4 patches to the current origami in the grid
                // indices are derived from the initial origami design, we want to copy around
                connect(s+8898 , n+6628);
                connect(s+10542, n+9510);
                connect(s+8350 , n+8962);
                connect(s+7802 , n+12904);
            }
        }
    }
}
```

Listing S14. Creating a 3x3 primitive cubic lattice from a single DNA origami (JS code)

SIV. SETTING UP ANM-OXDNA SIMULATIONS

A. Compiling ANM-oxDNA

ANM-oxDNA is an extension of oxDNA developed in C++ with CUDA acceleration. As such it requires a C++ compiler and CUDA installation. The recommended compiler for this project is gcc version 4-6. Due to some C++11 features being used in the older code, the allowed gcc version is very specific. Supported CUDA installations must be version 5-9. Once these prerequisites are satisfied you can compile the code by following the example procedure below:

```
git clone https://github.com/sulcgroup/anm-oxdna
cd oxdna
mkdir build
cd build
cmake ..    #Optional argument -DCUDA=1 for CUDA support
make -j6    #The integer after j is the number of threads to use to compile
```

Listing S15. Downloading and compiling the anm-oxDNA model (Shell)

B. Protein Parameterization with Python Scripts

Direct parameterization of a protein into our anm-oxDNA model requires solving for the pseudo-Inverse of the Hessian matrix to calculate the root mean squared fluctuations of each residue in the protein. Due to the amount of memory required and complexity of the calculation, oxView can only support parameterization of proteins consisting of less than 1000 residues. For larger proteins, python scripts are provided at `https://github.com/sulcgroup/anm-oxdna` in the /ANMUtils directory to perform the parameterization of the ANM and convert the system into oxDNA/oxView compatible files.
Below is an example script to demonstrate the Python module's intended usage:

```
# First we import the python module named 'models.py'
# Make sure you are in the correct directory!

import models as m

pdbfile = '~/Desktop/tmp/myprotein.pdb'

# The first function call should always be to read in information
# from the PDB file via the function get_pdb_info()

experimental_bfactors, xyz_coordinates = get_pdb_info(pdbfile)

# The optional parameter returntype in get_pdb_info can be altered to return information
# including sequence, chain ids, rigid body side chain vectors and other pdb information
# See the examples in the anm-oxDNA /ANMUtils directory for detailed usage

# Now we initialize a model using just the coordinates and B factors

our_anm = m.ANM(xyz_coordinates, experimental_bfactors, T=300, cutoff=13)

# T is temp in Kelvin at which the protein B factors were determined
# Cutoff is the Edge Cutoff value, once again in Angstroms
# As a guideline, i usually start with 12-13 A and will go up
# as high as 18A

# !Cutoff values must be set upon initialization, however creating
# several models at different cutoff values and comparing is easy!

# The ANM class serves as a base for the other classes
# which all others (except peptide) inherit from

# In all classes (except peptide) there is a one-shot function that
# automatically evaluates the analytical B-factors for that model.

#For our example ANM

```

```
our_anm.calc_ANM_unitary()

# The above function does the following:
# 1) Evaluates B-factors via SVD of Hessian
# 2) Automatically fits the spring constant to best fit with Experimental B-factors
# 3) Optional 'cuda' parameter is recommended for large structures (See /ANMUtils)

# Using Matplotlib we can plot our calculated B factors vs. the Experimental B factors

our_anm.anm_compare_bfactors(bfactor_comparison_image)

# saves figure of compared B factors to filepath bfactor_comparison_image

# Alternatively, for a more interactive experience Jupyter Notebook can also be used
# See /ANMUtils Setup example for more information on setting up a kernel
# and further usage in Jupyter notebook

# Assuming the B factors match close enough for your system of interest
# We now export our solved network into our oxDNA/oxView simulation files
# For all models, this consists of a single function call that uses the model itself
# as its main argument

m.export_to_simulation(our_anm, pdbfile)

# Generates Topology, Configuration, and Parameter File for system
```

Listing S16. Setting up a protein ANM model using Python (Python code)

SV. TRAJECTORY ANALYSIS WITH PYTHON

OxDNA trajectory files are plaintext files that, for an origami-sized structure, are often on the order of 10 GB in size. As such, it is important to have optimized algorithms and data structures that facilitate asking questions of your data set. Here, we introduce writing custom scripts using the tools provided in oxDNA Analysis Tools as a scaffold which makes working with these files as easy as possible.

In general, all scripts in oxDNA Analysis Tools follow the same general scaffold:

1. Define functions which compute the properties of interest for a single configuration. This can be pure Python or it can call outside programs such as DNAnalysis.

2. `if __name__ == "__main__":`

3. Parse command line arguments. We use the argparse library for this, however this is a stylistic choice that has no impact on the rest of the code. In general the following conventions are used for flags in the package:

   -h – Display the help and available command line arguments

   -p – Parallelize the computation oover the provided number of cpu cores

   -o – Primary output file name (if not from the required arguments)

   -d – Data file output name. In the package these data files are in the json format for oxView order parameter visualization.

   -v – Visualization file output name

   -i – Index file which tells the script to only run on a subset of the structure

   -c – Call the clustering algorithm when the computation is complete

   There are, of course, exceptions and additional flags for specific scripts, but these are general conventions that we try to follow when developing scripts.

4. Create a file reader object. There are two file readers available in the library:

LorenzoReader2 – Takes a trajectory and a topology file and recreates a similar object to the internal oxDNA representation. This is based on the data structures found in base.py from the main oxDNA distribution.

ErikReader – Takes only a trajectory file and creates an object with Numpy arrays for positions and the two orientation vectors. This is much faster than the LorenzoReader, however can only be used for applications where the identity of the parent strand of an individual nucleotide is unnecessary.

5. Call the previously-defined analysis function using the reader as an argument. This can be done using one of the parallelization functions which are available as part of the library. There are two different methods available for splitting the trajectory file:

One file – Attach multiple Python file readers to the trajectory file. On some systems, this can result in a significant slowdown of the analysis, which is why the second option is available.

Multi-file – Split the trajectory file into multiple temporary files and attach a single reader to each temporary file. This requires additional disk space and can leave junk files if the script fails before the cleanup step.

6. If a parallelization scheme was used, parse the data from each process and merge the data structures into the same format as if it came from a single process

7. Perform any post-processing steps on the full trajectory worth of data

8. Create plots, output files, and data summaries. We use Matplotlib for generating plots and generally use JSON encoding for our output files which often interface with one of the data overlay options in oxView. Many of the plots created by the library as relatively simple and we highly encourage users to edit the plotting function to match their own stylistic and visualization needs.

When writing your own analysis scripts, we highly recommend referencing the scripts in the package, particularly compute_mean.py and duplex_angle _finder.py as examples of using the ErikReader and LorenzoReader2, respectively.

As a shorter example than the aforementioned scripts, listing S17 is a simple script which calculates the dot products of the a1 orientation vectors for bonded nucleotides which was used in parameterizing oxView's base pair finder:

```
# import the libraries needed for this computation
# I added a symlink to oxdna_analysis_tools in a directory on my Python installation's PATH so I could import it like this.
# You could also write this script inside the oxdna_analysis_tools directory or add the path to your PYTHONPATH system variable to make the functions available for import.
import numpy as np
from oxdna_analysis_tools.UTILS.readers import LorenzoReader2
from oxdna_analysis_tools import output_bonds

# Define a function that calculates my quantity of interest.
# In this example, its the dot products of the a1 orientation vectors.
def get_orientation(mysystem, inp):
    # This function uses DNAnalysis to calculate the interaction energies between the particles in the system.
    energies = output_bonds.output_bonds(inp, mysystem)

    # These methods of the system object assigns each nucleotide to a strand and provides a reference to its paired nucleotide if one exists.
    mysystem.map_nucleotides_to_strands()
    mysystem.read_H_bonds_output_bonds(energies)

    # Create a list of a1 vectors of interacting nucleotides
    v1s = []
    v2s = []
    for s in mysystem._strands:
            for n in s._nucleotides:
                if len(n.interactions) == 1:
                    v1s.append(n._a1)
                    v2s.append(mysystem._nucleotides[n.interactions[0]]._a1)

    # Convert the list of a1 vectors to a Numpy array, which allows for faster computation.
    v1s = np.array(v1s)
    v2s = np.array(v2s)

    # Einstein summation is a compact matrix math representation which is faster than using for loops or broadcasting, both of which would also work here.
    dots = np.einsum('ij,ij->i', v1s, v2s)

    return dots
```

```
35
36 if __name__ == "__main__":
37     # Use Argparse to retrieve command line arguments
38     import argparse
39     parser = argparse.ArgumentParser(description="Calculates the dot product between a1 vectors of
       bonded nucleotides")
40     parser.add_argument('trajectory', type=str, nargs=1, help="The trajectory file to analyze")
41     parser.add_argument('topology', type=str, nargs=1, help="The topology file corresponding to the
        trajectory")
42     parser.add_argument('input', type=str, nargs=1, help="the input file used to run the simulation
       ")
43     args = parser.parse_args()
44
45     top_file  = args.topology[0]
46     traj_file = args.trajectory[0]
47     inp = args.input[0]
48
49     # Create a reader object
50     myreader = LorenzoReader2(traj_file,top_file)
51     mysystem = myreader._get_system()
52     all_orientations = []
53
54     # This was a simple program not intended for re-use so it was not parallelized.  Please see
       compute_mean.py for a good example of how to set up parallelization.
55     # Iterate through the trajectory and compute the quantity of interest for each configuration.
56     while mysystem:
57         print(mysystem._time)
58         all_orientations.extend(get_orientation(mysystem, inp))
59         mysystem = myreader._get_system()
60
61     # The goal of this exercise was to find out how far from directly antiparallel the vectors
       could be for oxDNA to still consider them paired
62     print(max(all_orientations))
63
64     # Create a simple plot to visualize the result
65     import matplotlib.pyplot as plt
66     bins = np.linspace(-1, max(all_orientations), 60)
67
68     plt.hist(all_orientations, bins)
69     plt.ylabel('count')
70     plt.xlabel('dot product')
71     plt.show()
```

Listing S17. An example of extending oxDNA Analysis Tools to compute an order parameter (angle between bonded nucleotides). (Python code)

You could then call this from a command line with:

```
Python3 orientations.py trajectory.dat topology.top
```

This will work with any set of trajectory/topology/input file you may have.