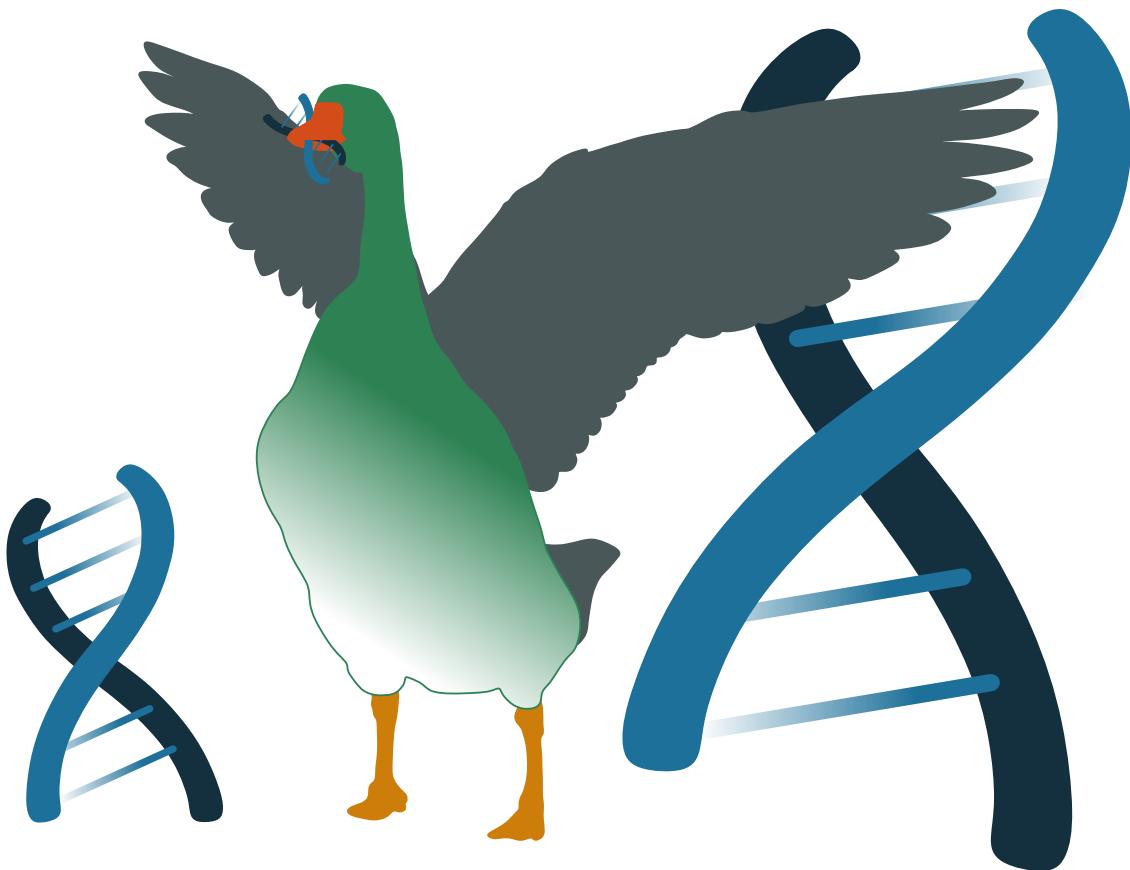


The Ducque Manual

Jérôme Rihon^{1,δ}

¹KU Leuven, Rega Institute for Medical Research, Medicinal Chemistry, Herestraat 49 - Box 1041, 3000
Leuven, Belgium *
^δ*Creator, Maintainer*



*All rights reserved to the Laboratory of Medicinal Chemistry Rega Institute of Medical Research Herestraat 49, 3000 Leuven, Belgium. Katholieke Universiteit Leuven (KUL)

Contents

1	Introduction	4
1.1	Modules	4
1.2	Package structure	5
2	Installing and running Ducque	6
2.1	Installation	6
2.2	Environment	6
3	Usage	7
3.1	Build	7
3.1.1	--sequence	7
3.1.2	--complement	7
3.1.3	--pdbname	7
3.2	Transmute	8
3.2.1	--pdb	8
3.2.2	--chemistry	8
3.2.3	--conformation	8
3.2.4	--moiety	8
3.2.5	--nucleobase	9
3.2.6	--bondangles	9
3.2.7	--dihedrals	9
3.3	Randomise	11
3.3.1	--sequence	11
3.3.2	--length	11
3.3.3	--chemistry	11
3.3.4	--complement	11
3.4	XYZ → PDB	12
3.4.1	--xyz	12
3.4.2	--residue	12
3.4.3	--atomname_list	12
3.5	Graphical User Interface (GUI)	13
3.5.1	Build	13
3.5.2	Transmute	14
3.5.3	Randomise	15
3.6	Available input by default	16
4	User implementation of customised nucleic acids	17
4.1	How to add to Ducque's src files.	17
4.1.1	Adding a nucleoside-linker combination to form a defined nucleotide	17
4.1.2	Adding one ore more conformations of a nucleoside chemistry	18
4.1.3	Adding the backbone atoms by which Ducque needs to build	19
4.2	Defining the correct torsion angle for the given nucleoside	20
References		22

Supervisor : prof. dr. Eveline Lescrinier

Co-promotor : prof. dr. Vitor Bernardes Pinheiro

Co-promotor : prof. dr. Matheus Froeyen

dr. Rinaldo Wander Montalvao for his guidance on the fundamentals of linear algebra.
dr. Charles-Alexandre Mattelaer for his guidance on Quantum Mechanics and without his
experimental work, Ducque could have never been conceived.

1 Introduction

There are no mistakes, only happy little accidents.

Bob Ross

A software for the purpose of building native and synthetic nucleic acid duplexes.

Ducque (IPA : /dʌk/) stands for :

D Acronyms

U Are

C Rather

Q

U

E Tedium



The name was inspired much in the same way that the ORCA Quantum Mechanics package^{1,2} was named. At the time of development, I had different name for Ducque. During one of my evaluation moments, one of the members of my advisory committee pointed out that with that name, I would probably not get enough traction, since Google searches gives millions of results. Given that fact, I tried coming up with a new name for the model builder. Weeks later, as I was working, my hat that was resting on my desk in front of me caught my eye. You see, my hat has a tiny duck embroidered on the front. With this idea, I "researched" ways to rewrite the word and get as little search engine results as possible, without degenerating the word itself and keeping it phonetically somewhat correct. Thusly, Ducque was born (a second time).

1.1 Modules

Ducque has five modules the user can access.

1. The `build` module will build the user the duplex structures we want to use for further experiments.
2. The `transmute` will be used to convert a specific nucleotide to a suitable input for Ducque to use as a `building block`.
3. The `randomise` module, will generate a randomised sequence that can be readily prompted to the `build module` to make a structure.
4. The `xyz_pdb` module exists because the many QM packages output `xyz`-formatted files after geometry optimisation. For most intents and purposes, the user might need to use `pdb`-formatted files. This module facilitates conversion from one format to the other through the CLI.
5. The `GUI` (Graphical User Interface) provides the user with clickable objects and predefined widgets to use Ducque, allowing the user to opt out of the CLI functionality. This module gives the user access to the `build`, `transmute` and the `randomise` module.

1.2 Package structure

path/to/Ducque

- **bin/**: contains executable to run Ducque.
- **docs/**: contains this pdf.
- **ff/**: contains the pdf on [*How_To_Build_A_Forcefield.pdf*](#).
- **json/**: contains building blocks requires by Ducque to build duplex sequences; json format.
- **src/**: contains Ducque's source code.
- **transmute/**: contains the transmute input files; tinp format.
- **xyz/**: contains the xyz_pdb input files.
- LICENSE (MIT).
- README.md (GitHub).
- setup.sh : required to make Ducque executable.

2 Installing and running Ducque

2.1 Installation

The installation is done by cloning the repository, through the use of [git](#). It is advised to clone the repository to the desired location at the [\\$HOME](#)-level ([\\$HOME](#), [\\$HOME/Desktop](#), ...).

Enter the Ducque directory and run the [setup.sh](#) script to give the correct permissions to the Ducque executable, located in [Ducque/bin](#).

Finally, add the last line to your bash environment, by adding it to your [\\$HOME/.bashrc](#).

For future reference, the [path/to/program](#) is the location on your system where Ducque is installed. E.g., if Ducque is installed at [\\$HOME/Desktop/Ducque](#), then [\\$HOME/Desktop = path/to/program](#).

```
# Install on your system
$ git clone https://www.github.com/jrihon/Ducque.git # clone repo locally
$ cd Ducque

# Run the setup.sh script
$ chmod a+x setup.sh      # Give permission to run as an executable script
$ ./setup.sh

# Add to ~/.bashrc
export PATH="$PATH:path/to/program/Ducque/bin"
```

2.2 Environment

To install the correct packages for Ducque, it is advised to use the [conda](#) package manager for Python modules. [Install conda](#).

To use Ducque, it is also advised to create a separate virtual environment so as to not override or break any dependencies in the base environment.

The user must [activate](#) the Ducque environment before installing the dependencies. The packages to install are [NumPy](#), [SciPy](#) and [Tkinter](#).

```
# Create a virtual environment using conda
$ conda create --name Ducque
# Enter the Ducque `conda environment`
$ conda activate Ducque

# Using the conda package manager install : NumPy, SciPy, Tkinter
$ conda install -c numpy scipy tk

# To leave the virtual environment
$ conda deactivate Ducque
```

To leave the virtual environment, the user can [deactivate](#) the conda environment to leave the [Ducque environment](#).

3 Usage

3.1 Build

The [Build](#) module allows to build a structure with a simple query. The build inputfile are highlighted with the `.binp` file extensions ([build-input](#)).

```
$ Ducque --build INPUTFILE.binp
```

To allow for high customisability of model duplexes, the user can build the following way :

- Leading and complementary strand with the same chemistry (homoduplex) (Fig. 1A.)
- A difference in chemistry between the in both strands (heteroduplex) (Fig. 1B.)
- Complementary strand with various chemistries (custom antisense strands) (Fig. 1C.)

A.
--sequence DT, DC, DA, DA, DC, DG, DG, DT, DA
--complement HOMO
--pdbname dna_homo_duplex

B.
--sequence RA, RG, RU, RU, RG, RC, RC, RA, RU
--complement MNA
--pdbname rna_mna_heteroduplex

C.
--sequence RC, RG, RA, RA, RG, RC, RC, RA
--complement RG, RC, 2MU, 2MU, RC, RG, RG, 2MU
--pdbname rna_2ome-rna_heteroduplex

Figure 1: Example of possibilities of queries accepted by Ducque.

3.1.1 --sequence

A list of comma-separated values. The values are specific nucleic acids that are implemented into Ducque. A list of values readily available to the user is presented in Sec. 3.6.

3.1.2 --complement

This can be one of three different types of queries.

1. The [HOMO](#) input, to denote that the complementary strand should be built with the same chemistry as the leading strand. The complementary sequence is inferred from [sequence](#) query.
2. The specific chemistry for the complementary sequence, such as [DNA](#), [RNA](#), [MNA](#) etc. The complementary sequence is inferred from the [sequence](#) query.
3. A list of comma-separated values, as shown in Sec. 3.1.1. A list of values readily available to the user is presented in Sec. 3.6.

3.1.3 --pdbname

The name of the [output](#) pdb-file. This can be appended by the `.pdb` extension, but is not required.

3.2 Transmute

The `transmute` module allows the user to incorporate nucleotides (building blocks) into Ducque's repository to be able to build nucleic acid duplexes with. The transmute inputfile are highlighted with the `.tinp` file extensions (`transmute-input`). The conversion to a suitable inputfile is stored as a `json`-formatted file in the repository itself. The json files are never to be written by hand and are mandated to be generated by the `transmute` module!

```
$ Ducque --transmute INPUTFILE.tinp
```

An example is given of the `tinp` file of DNA Adenosine :

```
--pdb dna_adenosine_2endo.pdb
--chemistry DNA
--conformation 2endo
--moiety nucleoside
--nucleobase A
--dihedrals -39.246, -151.431, 30.929, 156.517, 159.171, -98.922, -99.315
--bondangles 101.407, 118.980, 110.017, 115.788, 111.943, 119.045, 126.013
```

3.2.1 --pdb

Relative pathing to the `pdb` file of the molecule that the user wants to incorporate into the library. **Nota bene:** before using the `pdb` file, it has to be presented as an uncapped molecule. For example, when looking at Fig. 4, we see the HO5' and HO3' atoms capping the backbone. These atoms need to be removed before using the specific `pdb` file as an input for the `transmute` module! When finishing the building, terminal hydrogens are appended to either ends of the strands and their atomnames are inferred from the atom that bind the capping hydrogens.

3.2.2 --chemistry

The `chemistry` denotes the type of modification the molecule bears. For example: the standard deoxyribose nucleic acids are represented by adenosine, cytidine, guanosine and thymidine nucleosides, but are categorised under the `DNA chemistry`. The chemistry has direct relations to its `key` in Sec. 4.1.3.

3.2.3 --conformation

Ducque allows for multiple `conformations` to be built into the `complementary strand` of the generated duplex. For example: DNA chemistries are commonly determined in both the ${}^2'E$ and ${}^3'E$ configuration. Ducque allows that both are incorporated into the library (Sec. 4.1.2). The specific `conformation` is later inserted in the name of the `json` file to differentiate multiple conformers of the same chemistry and to avoid naming clashes in the repository.

3.2.4 --moiety

The Ducque software distinguishes two main part of molecule, being the `nucleoside` and the `linker`. Specifically, the building start with a linkerless nucleoside, to where to linker part is fitted on. Because

many different chemistries can share the same linker, and because it is useful to separate both parts when building and also on a coding-level, it was decided to allow nucleosides to be incorporated separately from the linker. Also, this diminishes boilerplate inputs. In Sec. 4.1.1, the user is mandated to couple the [nucleoside chemistry](#) and the type of [linker](#) together.

3.2.5 --nucleobase

The [nucleobase](#) flag is required to allow more flexibility in defining the [residue](#) name of the molecule. An earlier consideration was to infer the type of nucleobase from the residue name itself, but considering that we want to implement a method to accept any type of modified nucleobase in the future, the maintainer felt that this option might be more useful on a coding level and would disallow errors to propagate based on assumptions.

This flag represents the type of [nucleobase](#) that pertains to the [nucleoside](#) that the user wants to implement into Ducque. The [nucleobase](#) infers the complementary base with which it pairs; e.g. the [G](#) nucleobase infers basepairs with [C](#).

For now, only purine and pyrimidine nucleobases are implementable. Technically, if the nucleobase modification entails only additional substituents and the naming of the atoms in the ring system remains identical to that of the natural bases, then it should also be implementable. This feature has not been tested.

3.2.6 --bondangles

The [bondangles](#) that define the backbone of the nucleotide.

The way the duplex is built is from bottom to top (e.g. from 3' → 5' for DNA). The way this is reflected, among other things, is in the way the bondangles are required to be queried from the [json](#) files. In a set of a dihedral, meaning four (4) atoms, the latter three (3) are to be supplied as the bondangle of the respective backbone angle.

For example: the [γ-dihedral](#) contains the following set [C3', C4', C5', O5']. This means that the [γ-bondangle](#) contains the set [C4', C5', O5'] (Fig. 2). These values tend to be relatively stable across all different types of chemistries, given all atoms in a set are sp^3 hybridised. Therefore, one can use the same angles, pertaining to similar atoms involved in the bondangle, from other types of chemistries and it will build fine. For most, if not all, intents and purposes, the set of angles tend to remain relatively conserved. This means that, when implementing a new chemistry, the user should focus on the [dihedrals](#).

IUPAC nomenclature states the backbone follows the Greek alphabet : $[\alpha, \beta, \gamma, \delta, \epsilon, \zeta]$.^{3,4} This also involves the glycosidic dihedral of $[\chi]$ (Fig. 2A.). Since IUPAC follows the ordering of the alphabet, an additional backbone angle is added in case the type of chemistry requires an additional angle to be queried : $[\eta, (\epsilon\tau\alpha)]$.

Not all chemistries require all stated backbone angles and thus the software permits shorter backbone chains too, like PeptideNA and ThreoseNA, to be built all the same.

To emphasize : The angles given by the user to the building block are in fact the angles Ducque will use in order to build out the molecular structure! Giving erroneous bondangle values will result in warped molecular structures.

3.2.7 --dihedrals

The dihedral (torsion angles) that define the backbone of the nucleotide.

A dihedral constitutes an angle between four atoms. For example: the γ -dihedral contains the set

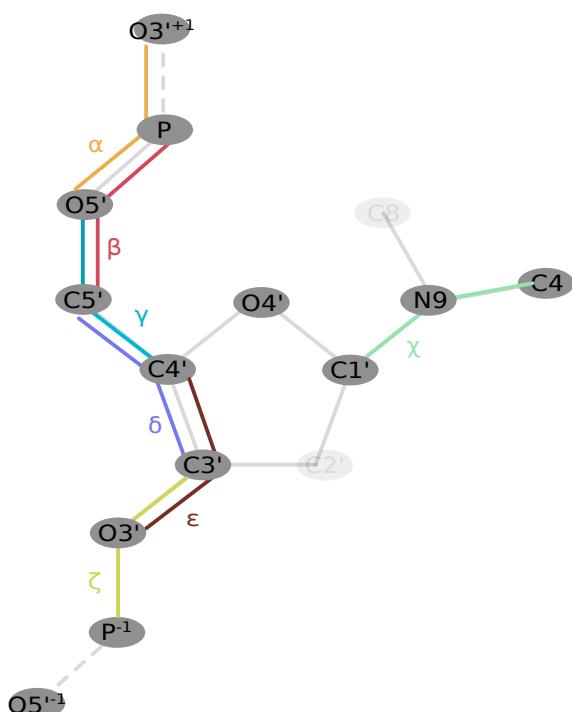
[C3', C4', C5', O5']. To procure these values, we redirect the reader to Sec. 4.2 of this manual for more information.

IUPAC nomenclature states this follows the Greek alphabet : $[\alpha, \beta, \gamma, \delta, \epsilon, \zeta]$.^{3,4} This also involves the glycosidic angle of $[\chi]$ (Fig. 2B.). Since IUPAC follows the ordering of the alphabet, an additional backbone torsion is added in case the type of chemistry requires an additional angle to be queried : $[\eta, (\text{eta})]$.

Not all chemistries require all stated backbone dihedrals and thus the software permits shorter backbone chains too, like PeptideNA and ThreoseNA, to be built all the same.

To emphasize : The dihedrals and angles given by the user to the building block are in fact the dihedral/angle Ducque will use in order to build out the molecular structure! Given erroneous dihedral values will result in *warped* molecular structures.

A. BOND ANGLE ANNOTATION



B. DIHEDRAL ANNOTATION

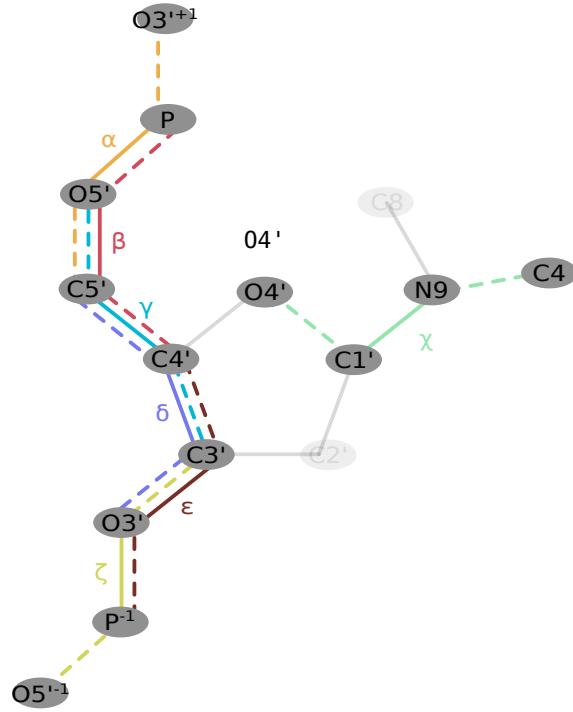


Figure 2: Representation of the **A.** bond angles and **B.** dihedrals in the backbone of nucleic acid nucleotides. This represents how one can consider inputting the values for their own type of chemistries. Notice the bottom to top direction in the annotated bond angles (**A.**).

3.3 Randomise

This module is implemented to facilitate writing and producing sequences. The `randomisation` inputfile are highlighted with the `.rinp` file extensions (`randomise input`).

```
$ Ducque --randomise INPUTFILE.rinp
```

- A variable sequence with a specified chemistry (Fig. 3.A).
- A way to prepend a given sequence with a prompted chemistry. This method is not a randomisation, but allows the user to write the desired sequence faster. This functionality was more fit to be introduced in this module than anywhere else (Fig. 3.B).

This query outputs a `binp` file that can be readily passed to the build module (Sec. 3.1).

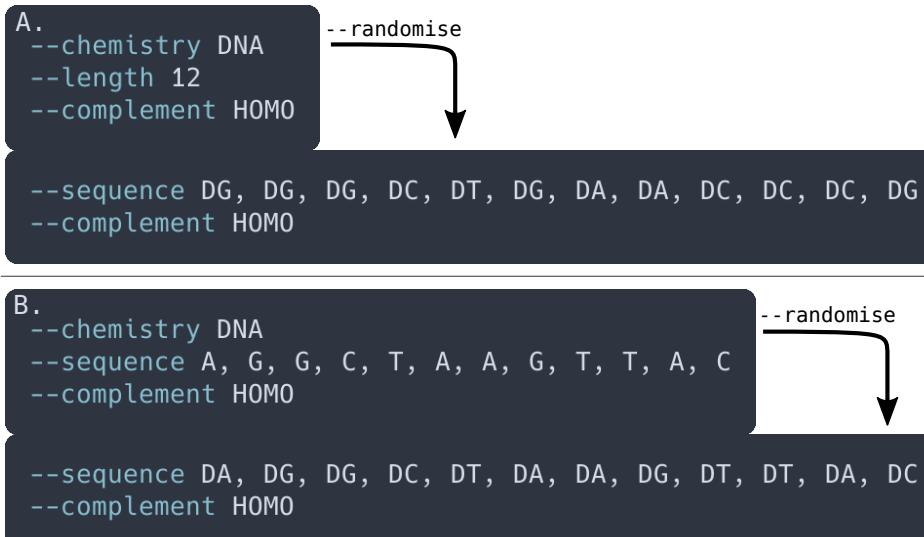


Figure 3: Examples of queries for the `randomisation` module.

3.3.1 --sequence

A mutually exclusive flag with `--length`. If a given sequence is given, then the length of it is already determined by the user. The sequence is a list of comma-separated values, consisting of one of five currently implemented nucleobases : `A`, `C`, `G`, `T`, `U`.

3.3.2 --length

A mutually exclusive flag with `--sequence`. This requires an `integer` argument to be passed into. It returns a randomised sequence, for a given chemistry, of the leading strand with a set `length`.

3.3.3 --chemistry

Prompt a `chemistry type`, from a list, to query Ducque for a correctly typed sequence. The type of chemistry is related to the `keys` of the `entries` given in Sec. 4.1.3.

3.3.4 --complement

The `complement` argument can be filled in at the `randomisation` stage, but its sole function is to pass the argument to the build module later. This is an optional argument.

3.4 XYZ → PDB

The `xyz_pdb` module allows the user to convert from an `xyz` to an `pdb` formatted file. Both are generally readable by all Molecular Viewing Software, but the `pdb` format is the only format accepted by Ducque, in order to `transmute` the molecule into a suitable building block for model building. An `xyz` file is a file consisting only of the atomic coordinates and the atomic elements.

```
$ Ducque --xyz_pdb INPUTFILE
```

```
--xyz dna_adenosine_2endo.xyz
--residue DA
--atomname_list HO5', O5', C5', H5'1, H5'2, C4', H4', O4', C1', ..., O3', H03'
```

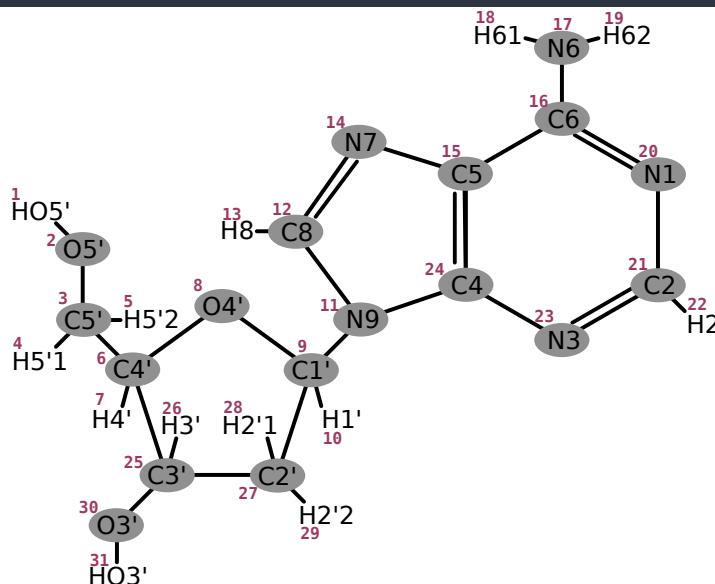


Figure 4: Depiction of the way in which atomnames are assigned according to the (AMBER Tree Structure).

3.4.1 --xyz

Relative path to the `xyz` file the user wants to convert.

3.4.2 --residue

The name of `residue` attributed to the specific chemistry, as per the [PDB format](#). String names can not exceed three (3) characters. The residue name has direct relations to its `key` in Sec. 4.1.1 and Sec. 4.1.2.

3.4.3 --atomname_list

The `atom name` attributed to the specific atom in the molecule, as per the [PDB format](#). String names can not exceed four (4) characters per atom name. Fig. 4 follows the AMBER Tree Structure.

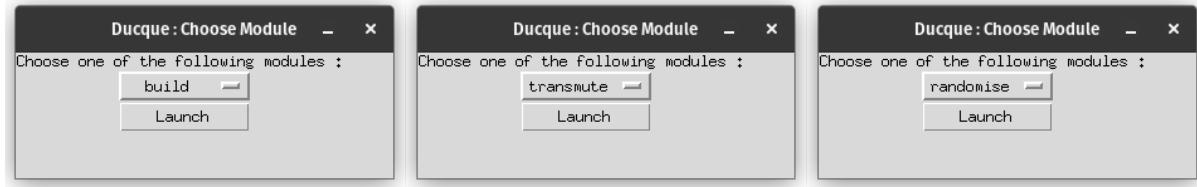
3.5 Graphical User Interface (GUI)

The `GUI`-module allows the user to access the other modules from a neat GUI instead of the CLI. The GUI is called by the following command, and specific modules are more quickly accessed by adding to the query :

```
# Ducque calls to GUI modules
$ Ducque --gui

# Query to open specific modules
$ Ducque --gui build
$ Ducque --gui transmute
$ Ducque --gui randomise
```

The first command gives the user the option to acces the modules, this defaults on `build`.



The `xyz_pdb` module was not made available as a GUI because it was difficult to implement a neat interface. As such, it is only available as a CLI command, where it performs better.

3.5.1 Build

We refer to Sec. 3.1 for the explanation on the various `--build` arguments.

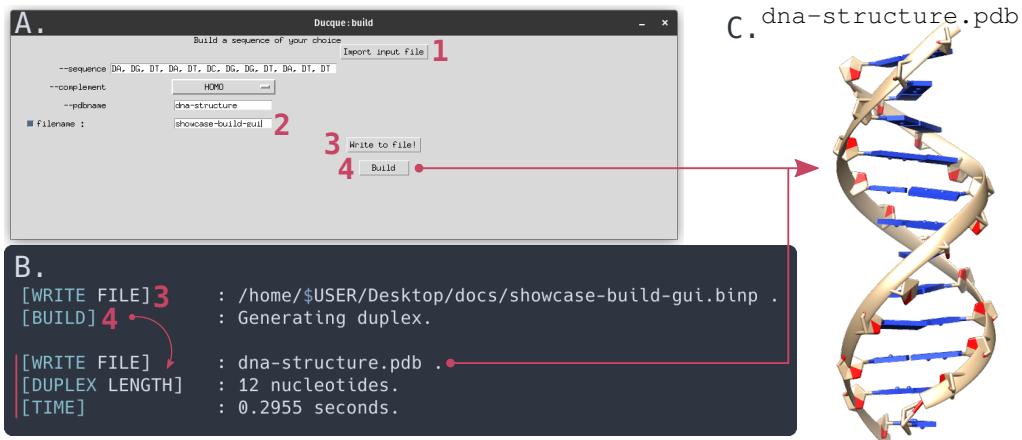


Figure 5: **A.** The `build` GUI. **B.** What gets outputted to `stdout` when using Ducque. **C.** The produced molecular structure.

1. Import a `binp` if desired or if possible. The user can also just write the query by hand.
2. Optionally, rename the `binp` file to something else.
3. Save the new `binp`, after having filled in the `pdbname` query.
4. Build a model structure based on the produced `binp` file.

In Fig. 5A., we see the GUI of this module. The `complement` flag show a option-list from which we can query the type of chemistry we want. This list is dependent on the `keys` of the `table` defined in Sec. 4.1.3.

3.5.2 Transmute

We refer to Sec. 3.2 for the explanation on the different arguments.

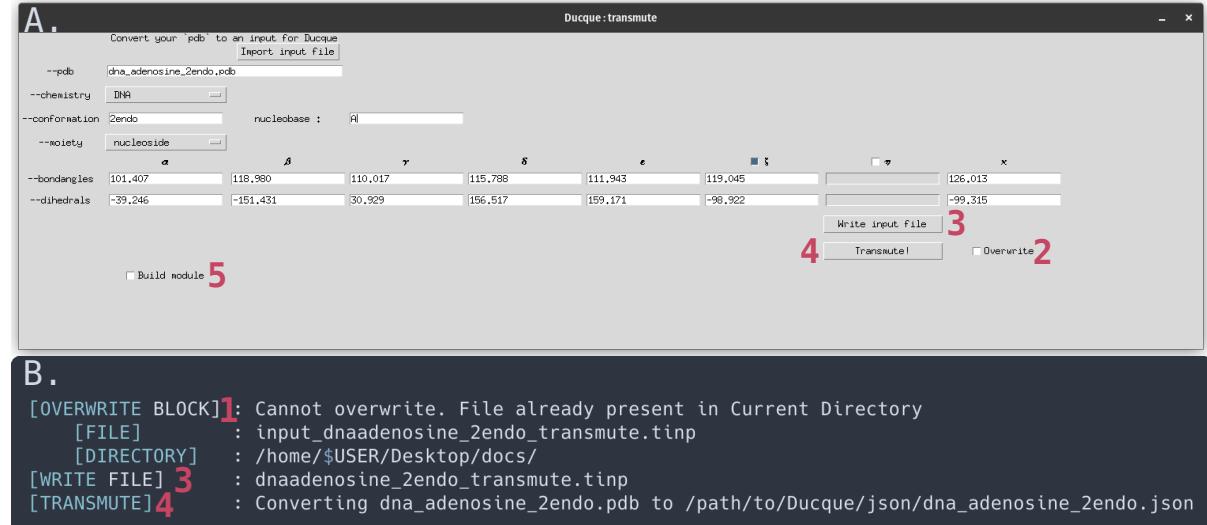


Figure 6: **A.** The `transmute` GUI. **B.** What gets outputted to `stdout` when using Ducque.

- After having imported a specific `tinp` file and modified some of the data, we can write to the same file. However, to avoid accidental overwriting of the same file, the user can only write to the same file in the current working directory if the `Overwrite` button has been checked. This also prohibits from using the `Transmute!` button, if the `json` already exists in the `path/to/Ducque/json` directory.
- Check the button to allow writing to existing files on the system. **3.** With the given query, a `tinp` file will be written to the current working directory. **4.** Convert the `tinp` to a `json` file. **5.** This functionality is a built-in part of the `build` module, within the `transmute` module. This will come in later into play, but essentially facilitates the iterative approach to tweaking and modifying torsion angles for the given input by allowing `transmuting` and `building` within a few clicks. More on this in Sec. 4.2.

In Fig. 6A., we see the GUI of this module. The `chemistry` flag show a option-list from which we can query the type of chemistry we want. This list is dependent on the `keys` of the `table` defined in Sec. 4.1.3.

3.5.3 Randomise

We refer to Sec. 3.3 for the explanation on the various `--randomise` arguments.

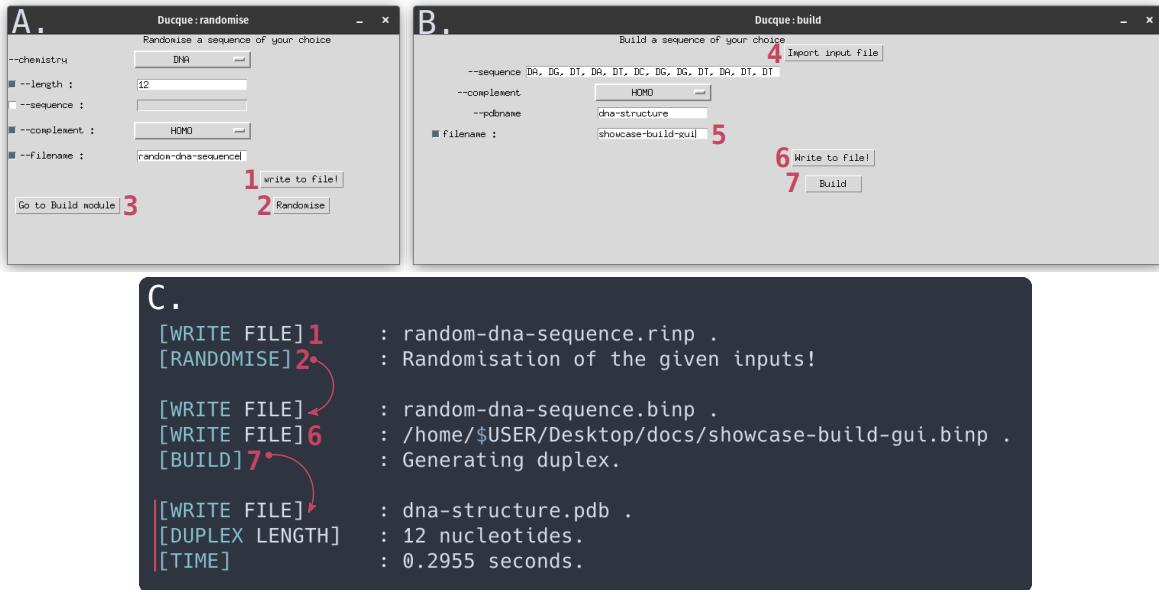


Figure 7: **A.** The `randomise` GUI. **B.** The `build` GUI. **C.** What gets outputted to `stdout` when using Ducque.

1. After filling in the query, write to a `rinp` file.
2. Randomise an output based on the produced `rinp` file. This produces a `binp` file, with which one can readily build.
3. Click the button to close the `randomisation` module and go straight to the `build` module.
4. Import the created `binp` into the module to fill in the queries.
5. Optionally, rename the `.binp` file to something else.
6. Save the new `binp`, after having filled in the `pdbname` query.
7. Build a model structure based on the produced `binp` file (Fig. 5C.).

In Fig. 7A., we see the GUI of this module. The `chemistry` and the `complement` flag show a option-list from which we can query the type of chemistry we want. This list is dependent on the `keys` of the `table` defined in Sec. 4.1.3.

3.6 Available input by default

Deoxyribose NA (DNA)	Ribose NA (RNA)	β -homo NA (B-HOMODNA)
DA DC	RA RC	DDA DDC
DG DT	RG RU	DDG DDT
+		
Hexitol NA (HNA)	Xylose NA (XYNA)	Deoxyxlose NA (DXYNA)
HA HC	XA XC	DXA DXC
HG HT	XG XU	DXG DXT
+		
2-O-Methyl RNA (2-OME-RNA)	Cyclohexenyl NA (CENA)	Morpholino NA (MNA)
2MA 2MC	CA CC	MA MC
2MG 2MU	CG CU	MG MT
+		

Figure 8: The headers contain the type of chemistry, with their abbreviation included in brackets. There are NINE (9) types of chemistries in total already implemented in Ducque.
A : Adenine, C : Cytosine, G : Guanine, T: Thymine, U: Uracil, NA: Nucleic Acid

4 User implementation of customised nucleic acids

4.1 How to add to Ducque's *src* files.

This section concerns with the incorporation of [custom types of chemistries](#) into [Ducque's repository](#). This is done by modified one of the [python](#) files in the [source code](#) itself.

```
# Go to the src files
$ cd path/to/program/Ducque/src/builder/

# Open file in your preferred text editor
$ vim builder_library.py
```

The reason why this a python script is modified, and not some sort of configuration file, is two-fold :

1. In order to create a configuration file, one needs to depend on various typed formats, like [yml](#), [json](#), [toml](#) etc. Though many libraries exist to parse these formats, they (especially the first two) tend to be somewhat finnicky if implemented by hand. To combat this issue, one would have to write a specified format and have it be parsed by such a library. Writing a specific format brings many I/O issues that bring more trouble than they are useful. When in reality, the only thing we'd need is a dictionary (a.k.a. table, hashmap ...) with the given inputs. This brings me to my second point.
2. What better parser can be asked to parse a native [data structure](#), the dictionary, than the [python interpreter](#) itself. What the [maintainer](#) proposes is that errors in [python source code](#) are caught immediately. This means that, upon running Ducque, all code is checked and looked for [syntax errors](#). If, and when, these errors are encountered, Ducque will not be called, which is the desired behaviour when such an error has been found. Even better, the [Python Error Handling](#) will return a message stating the exact problem of why the error has been encountered and thus one can fix their [syntactic mistakes](#). This often involves missing a comma, square brackets or curly braces.

4.1.1 Adding a nucleoside-linker combination to form a defined nucleotide

This section allows the user to implement the correct combination of [nucleoside](#) and [linker](#) to [Ducque's repository](#). The way this is formed is through [{KEY : VALUE}](#) pairs, more commonly referred to as a dictionary.

Important to note here is that the given conformation will be the conformation with which [Ducque](#) will [build to leading strand](#), for the nucleoside-linker pair in [TABLE_NUCLEOTIDES](#)!

```
# GO TO +- line 18
# -----
#           NUCLEOSIDE      ,      LINKER
# -----
DH = path/to/Ducque # DUCQUE_HOME
TABLE_NUCLEOTIDES = {
    "DA" : [ DH + "dna_adenosine_2endo.json", DH + "dna_phosphate.json" ],
    "DG" : [ DH + "dna_guanosine_2endo.json", DH + "dna_phosphate.json" ],
    #
    #           ...
    "FOO" : [ DH + "foo_nucleoside.json", DH + "bar_linker.json" ],
}
```

- The **KEY** is the [residue name](#) (Sec. 3.4.2) that was used in the [pdb](#) file (Sec. 3.2.1), upon transmuting the parameters of the desired nucleoside. This **KEY** is a [string](#) and has to be **ALL CAPS**.
- The **VALUE** is a python [list](#) that contains two values : the name of the nucleoside and the name of the associated linker, as [string](#)-typed values.
- The **DH** variable (DUCQUE_HOME) is the place where Ducque lives on the user's machine; this is evaluated at runtime. The user needs only to prepend '**DH +**' to the name of the files pertaining to the nucleotide incorporated. To clarify, the inputfiles for Ducque are stored as [json](#) files in the [path/to/Ducque/json/](#) directory, upon transmuting the inputs, so this removes a lot of boilerplate text in the repository.

4.1.2 Adding one ore more conformations of a nucleoside chemistry

This section is similar to the Sec. 4.1.1, but builds [the complementary strand](#). This allows the user to implement [multiple conformations of the same chemistry](#) to be used while building that strand. Building out the complement involves several fitting procedures. If more than one conformation exist of a type of chemistry, Ducque will try its best to fit the best conformation to the leading strand and its adjacent stacking nucleotides.

Again, the **{KEY : VALUE}** pairs are crucial to our repository.

```
# GO TO +- line 75
# -----
#      COMPLEMENTARY REPOSITORY
# -----
TABLE_CONFORMATIONS = {
    "DA": [ DH + "dna_adenosine_2endo.json", DH + "dna_adenosine_3endo.json"],
    "RG": [ DH + "rna_guanosine_3endo.json"],
    #           ...
    "FOO": [ DH + "foo_nucleoside_Xendo.json", DH + "foo_nucleoside_Yexo.json"],
}
```

Figure 9: A dictionary that is used to insert multiple conformations into Ducque for a specific residue, to build out the complementary strand.

- The **KEY** is the [residue name](#) (Sec. 3.4.2) that was used in the [pdb](#) file (Sec. 3.2.1), upon transmuting the parameters of the desired nucleoside. This **KEY** is a [string](#) and has to be **ALL CAPS**.
- The **VALUE** is a python [list](#) that contains two values : the name of the respective nucleoside conformations, as [string](#)-typed values.
- The **DH** variable (DUCQUE_HOME) is the place where Ducque lives on the user's machine; this is evaluated at runtime. The user needs only to prepend '**DH +**' to the name of the files pertaining to the nucleotide incorporated. To clarify, the inputfiles for Ducque are stored as [json](#) files in the [path/to/Ducque/json/](#) directory, upon transmuting the inputs, so this removes a lot of boilerplate text in the repository.

4.1.3 Adding the backbone atoms by which Ducque needs to build

As stated before (Sec. 3.2.6), the software builds from bottom to top (e.g. for DNA: $3' \rightarrow 5'$). This section defines the backbone by which the Ducque model builder should produce the structure with. In the example (Fig. 10), the DNA and MNA nucleic acid chemistries are given. Again, the {KEY : VALUE} pairs are crucial to our repository.

```
# GO TO +- line 130
# -----
#      BACBKBONE REPOSITORY
# -----
# Add atoms of the backbone in ASCENDING ORDER OF APPEARANCE!
TABLE_BACKBONE = {
    "DNA" : ["O3'", "C3'", "C4'", "C5'", "O5'"],
    "MNA" : ["N3'", "C4'", "C5'", "C6'", "O6'"],
    #
    ...
    "FOO" : ["Z", "Y", "X", "W", "V"],
}
```

Figure 10: The dictionary that is used by Ducque to build along a given backbone. One of the most crucial aspects to Ducque and why it allows any backbone modification to be built with this software.

- The KEY is the `chemistry` that was used in the `tinp` (Sec. 3.2.2), upon transmuting the parameters of the desired nucleoside. This KEY is a `string` and has to be `ALL CAPS`.
- The VALUE is a python `list` that contains all atoms of the backbone, in `ascending order` (callback to Sec. 3.2.6). The list is a set of comma-separated `string`-typed values, that pertain to the `atomnames` (Sec. 3.4.3) in the prompted `pdb` file (Sec. 3.2.1).

Fig. 10 refers to the storage of the information on the `DNA chemistry` ($3' \rightarrow 5'$) and the `MNA (Morpholino) chemistry` ($3' \rightarrow 6'$). This is why it is important to be `consistent` with the nomenclature of the atoms in the molecule and the order in which they appear in the list. This functionality is what gives Ducque its power, by being able to build with any type of `backbone`! Ducque will query its repository, based on the `user's input values`. If these do not align with each other, then Ducque will not be able to build the desired structure, so be precise with your inputs.

4.2 Defining the correct torsion angle for the given nucleoside

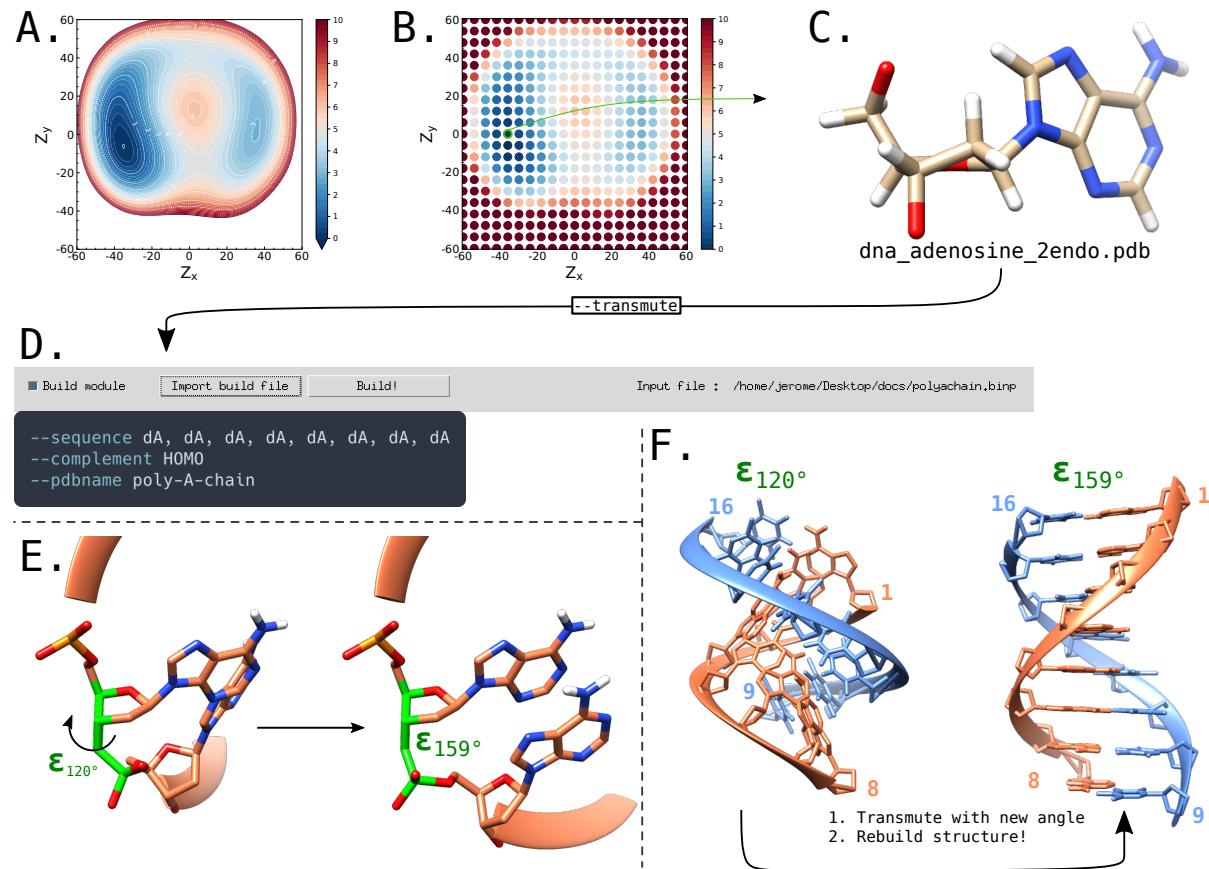


Figure 11: **A.** Potential Energy Surface (PES) of DNA Adenosine. **B.** A scatterplot that used to create the PES, by interpolating gridpoints. **C.** Uncapped DNA Adenosine, ready for conversion to json file. **D.** The build module, embedded into the transmute module. This makes for easy access to building new structures after modifying angle values. **E.** Closer look to how one can iteratively optimise backbone angle values. The complementary strand is not shown here to reduce visual noise. **F.** Leading strand in coral, complementary strand in cornflower. This exemplifies how the differences in backbone angle values can have a dramatic change on the resulting structure.

Fig. 11C. The user can curate the Conformational Sampling⁵ landscape for a specific conformation to be built into Ducque's repository. Here is a depiction of the DNA Adenosine nucleoside (${}^2'E$), prepared as an uncapped nucleoside, ready to be transmuted by Ducque.

In Fig. 11D., the embedded build module is shown, which is a callback Sec.3.5.2-5. Here, the user loads in a finished binp to the Import build file button. Whenever the user makes changes to the produced json library file, upon transmuting, the user just has to click the Build! button to produce a new structure, which will build a structure with the recently modified backbone angle values. This makes it possible for the user to iteratively adjust and build structures without having to leave the transmute GUI.

The author suggest the following protocol, as devising the correct backbone dihedrals can be quite difficult.

- It is advisable that, whenever curating conformers from the landscape (Fig.11B.), that the same conformer is picked for all nucleobase types. It has been noticed that purines and

pyrimidines tend to have similar backbone angle values respectively. This makes the model building simpler; the goal to a good structure is building a model that is fit for upcoming simulations.

- As shown in Fig.11E., the structure can start from a starting value, here arbitrarily chosen to be ϵ_{120° . By opening your preferred Molecular Viewing Software (UCSF Chimera, PyMol ...), one can select a dihedral and rotate around it, up to the point that the basepairs stack well with eachother. Emulating, or at least trying to emulate, stacking conditions is a good indicator of going into the right direction.
- Another suggestion is to use [poly-N sequences](#) to build the structure. This will make it far easier to modify angles of one type of nucleoside at one time.
- Note that in order to build a [poly-N structure](#), the [complement of N](#) needs to be implemented too! Though the complement does not need to have its angles optimised just yet, as base-pairing conditions are the more important factor in fitting the complementary strand when building the molecular structure.

References

- [1] Frank Neese, Frank Wennmohs, Ute Becker, and Christoph Riplinger. The ORCA quantum chemistry program package. *J. Chem. Phys.*, 152(22):224108, jun 2020.
- [2] Frank Neese. Software update: The ORCA program system—version 5.0. *WIREs Computational Molecular Science*, 12(5), mar 2022.
- [3] IUPAC-IUB Joint Commission on Biochemical Nomenclature (JCBN). Abbreviations and symbols for the description of conformations of polynucleotide chains. *European Journal of Biochemistry*, 131:9–15, 1983.
- [4] Wolfram Saenger. *Structures and Conformational Properties of Bases, Furanose Sugars, and Phosphate Groups*, pages 51–104. Springer New York, 1984.
- [5] Charles-Alexandre Mattelaer, Henri-Philippe Mattelaer, Jérôme Rihon, Matheus Froeyen, and Eveline Lescrinier. Efficient and accurate potential energy surfaces of puckering in sugar-modified nucleosides. *Journal of Chemical Theory and Computation*, 17(6):3814–3823, May 2021.