The Ducque Manual

Jérôme Rihon[1,δ]

[1]KU Leuven, Rega Institute for Medical Research, Medicinal Chemistry, Herestraat 49 - Box 1041, 3000 Leuven, Belgium *

[δ] *Corresponding author, Maintainer*

Contents

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 Tedious

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 gaves 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 embroided 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, which is the primary module for to use. This will build us the duplex structures we want to use for further
2. The **transmute** will be used, together with an elaborate input, to convert the ‘pdb‘ file to a suitable input for Ducque to use as a building block. The appropriate format has been settled to be the very simple ‘json‘ format.
3. For testing purposes or just to be able to generate randomised sequences, there is a **randomise** module, which generates a randomised duplex structure based on a set of given inputs.
4. To be able to convert ‘xyz‘ formatted molecule structures to a ‘pdb‘ format, I have included a **xyz_pdb** module. Since I am an avid ORCA user and the ‘xyz‘ format is often outputted, I wanted to make it more managable to format a ‘pdb‘ file for myself during development.
5. To have clickable objects for the standard non-terminal user, there is a simple **GUI** that can be used to employ the following modules : build, transmute and randomise. The reason that the xyz_pdb is not included is because I did not find a nice way to design a simple gui. Secondly, other QM programs might have different outputs of molecule structures, the ‘xyz‘ format can be rather niche and therefor not worth to effort to design a good gui for. Chimera and PyMol can definitely convert from one format to the other if one cannot work through the CLI.

1.2 Package structure

path/to/Ducque

→ **bin/:** contains executable to run Ducque.

→ **docs/:** contains this pdf and its LaTeX sourcefiles.

→ **json/:** contains building blocks requires by Ducque to build duplex sequences. The name for the json file is derived from the inputs of the `--transmute` INPUTFILE. The filename is there for classification and is thereby non-trivial.

→ **puckerdata/:** contains all optimised *xyz* and *pdb* files, needed by `--transmute` to be converted to *json*.

→ **src/:** contains all the Ducque source code.

→ **transmute/:** contains the `--transmute` input files, just as an example. The name of the file itself is trivial.

→ **xyz/:** contains the `--xyz_pdb` input files, just as an example. The name of the file itself is trivial.

→ LICENSE .

→ README.md .

→ setup.sh : required to make Ducque executable.

2 Usage

2.1 Introduction

All Ducque modules are callable from either the CommandLine Interface (CLI) or through the Graphic User Interface (GUI). The most prominent module is ofcourse the `--build` module, where one can easily write an inputfile to generate a nucleic acid model structure. Ducque can be used in scripting methods through the CLI, to generate a multitude of structure rapidly, and can also be used through the GUI. All GUI modules have a useful method of first writing a file to the system, before reading in that file and creating the desired output. This emulates a logging system, so that one can reuse inputfiles any time. All GUI modules (build, transmute, randomise) come equipped with a nifty method to read in an already available inputfile, to fill in all the fields that the prompt requires!

2.2 Graphical User Interface (GUI)

Ducque's GUI is called from the CLI first! Ducque had always been intended to be a CLI tool at heart, but it was later decided to build out an interface to gain a larger audience. Eventually, building new nucleotides into Ducque's library `transmute` module ended up being far more practical through the GUI.

The practicality of the GUI lies in that one does not have to remember the different options one has to use in order to use Ducque. With a template inputfile, one can script generating multiple files using basic shell and call Ducque through scripting, at least for the `randomise` and `build` modules. Ducque's GUI can be called as is or with options. The first call takes you to a selection of the different modules. The other call take you directly to the respective modules.

```
# Ducque calls to GUI modules
$ Ducque --gui
$ Ducque --gui build
$ Ducque --gui transmute
$ Ducque --gui randomise
```

2.3 Build

The **Build** module allows to build a structure with a simple query. The *sequence* and *complement* flags are mandated. The *pdbname* allows you to name the produced pdb structure. Defaults to the name of the prompted INPUTFILE.
The GUI module has an added flag *filename*, to which you need to prompt a filename, which is the INPUTFILE that is used to build the pdb structure. Defaults to *random_sequence*.
Small overview of the *build* module functionality

```
$ Ducque --build INPUTFILE
      The inputfile is read in and the sequence is built accordingly.
      ' * ' : mandatory
      ' + ' : additional in the GUI

INPUTFILE : [
--sequence SEQUENCE *
  Only valid input in the file just a string of nucleotides (comma-delimited).
      Example: --sequence   dT, dC, dA, dA, dC, dG, dG, dT, dA

--complement COMPLEMENT *
  The complement flag denotes the sequence of the complementary strand.
  A list of nucleotides is also a valid input (comma-delimited).
      Example: --complement homo
      Example: --complement rA, rG, rT, rT, rG, rC, rC, rA, rT

--pdbname PDBNAME
  To prompt the name of the pdbfile for the outputted structure.
  If none is given, this defaults to the name of the INPUTFILE.
        Example: --pdb foobar.pdb
]
GUI : [
--filename FILENAME +
  The name of the INPUTFILE to write to.
]
```

2.4 Transmute

The transmute is fully equipped to make a proper building block file for the Ducque software. The small overview here below gives a good idea of what is needed and a detailed explaination is given to complement it.

NB : The GUI module performs far better than the CLI version of this tool and therefore is highly recommended to only use the GUI module itself.

Small overview of the *transmute* module functionality

```
$ Ducque --transmute INPUTFILE
The inputfile is read in and the json file is formatted accordingly.
      ' * ' : mandatory
      ' + ' : additional in the GUI

INPUTFILE : [
--pdb PDB *
  The name of the file of the structure you want to convert to json
      Example: --pdb dna_A.pdb

--chemistry ID *
  The chemistry that defines the given nucleotide
      Example: --chemistry DNA

--comformation CONFORMATION *
  The conformation that denotes the nucleic acid.
   Allows multiple conformers for a given chemistry.
  Used to name the output json file.
  The complementary strand is then fitted onto the leading strand.
      Example: --conformation 2endo
      Example: --conformation 1-4boat

--moiety MOIETY *
  The moiety that the structure defines. Should either be "nucleoside" or "linker"
      Example: --moiety nucleoside

--bondangles ALPHA, BETA, GAMMA, DELTA, EPSILON, ZETA, CHI *
  The backbone angles, DEGREES (comma-delimited).
      Example: --bondangles 101.407, 118.980, 110.017, 115.788, 111.943, 119.045, 126.013

--dihedrals ALPHA, BETA, GAMMA, DELTA, EPSILON, ZETA, CHI *
  The backbone dihedrals, DEGREES (comma-delimited)
      Example: --dihedrals -39.246, -151.431, 30.929, 156.517, 159.171, -98.922, -99.315
]
GUI : [
  nucleobase +*
  This entry takes in of the {A,C,G,T,U} bases and assigns the
   correct nucleobase's name to the file.
]
```

- **pdb:** Relative pathing to the *pdb* in question is possible for the current working directory and any children directories thereof. Pathing up seems unnecessary.
- **chemistry:** This is there to relate the different nucleobases of a given chemistry with each other.
- **conformation:** Certain chemistries require multiple conformations to be built in, depending on the type of structure one would like to build. This feature allows to implement multiple different conformations of the same chemistry + nucleobase.
- **moiety:** Is either a nucleoside or a linker moiety. This is required because Ducque needs to know which type is added to the library to react accordingly.
- **bondangles:** all necessary bondangles of the backbone need to be added.
- **dihedrals:** all necessary dihedrals of the backbone need to be added.

- **nucleobase:** A GUI feature, but an important one. This asserts the correct nucleobase to the chemistry and avoids mistakens from happening. This is also the reason why we mandate the usage of `transmute` through the GUI and not the CLI.

2.5 Randomise

This module is implemented to generate strands of a variable sequence or to generate sequences of a combination of one chemistry with a specified sequence. In general, to make inputfiles for the `build` module more ergonomic.

It was decided to make the `--length` and the `--sequence` mutually exclusive. This, because one cannot specifiy a random sequence of a length of X basepairs and also prompt a specific sequence. So one uses the `--chemistry` flag with one or the other. The `--complement` option can be prompted as well. This query just gets passes to the building query as is, to ready the file as soon as the randomised sequence has been generated.

Small overview of the *randomise* module functionality

```
$ Ducque --randomise INPUTFILE
The inputfile is read in and --Ducque inputfile is generated.
NB : Using --length and --sequence are mutually exclusive.
      ' * ' : mandatory
      ' + ' : additional in the GUI
      '*| ' : and / or

INPUTFILE : [
  --chemistry CHEMISTRY *
  The chemistry that defines the prompted nucleotide's
      Example: --chemistry DNA
      Example: --chemistry MNA

  --length LENGTH *|
  The length, in amount of nucleotides, of the sequence the user wants to generate.
      Example: --length 30

  --sequence SEQUENCE *|
  Provide the file with a sequence. Only the bases are required.
  The values are comma-delimited :
      Example: --sequence A, C, T, G, G, A, A, T, C, A

  --complement COMPLEMENT +
  Fills in the `--complement` flag in the input file to gui
  Can be filled in with a sequence or a particular chemistry
      Example: --complement homo
      Example: --complement DNA
]
```

2.6 XYZ → PDB

This module exists for the sole reason of converting *xyz*-file formats to *pdb*-file formats. As the maintainer has always worked with ORCA, ORCA tends to output geometry optimised structures in as *xyz*. While one can open UCSF Chimera or PyMol, read in the file and save as *pdb* and adjust atom naming manually, it was felt that no outside programs had to be required to make formatting more manageable. (See pdb file format structure).

This module does not appear in the GUI part of the software, since designing a proper UI for it, while not getting too convoluted and ugly, was rather challenging. Because this module is of more practical use than anything else and not mandated for users to actually use, the GUI portion was scrapped.
For the inputs, several restrictions are put in place.

- **xyz:** Relative pathing to the *xyz* in question is possible for the current working directory and any children directories thereof. Pathing up seems unnecessary.
- **residue:** String names can not exceed THREE (3) characters, as per the *pdb*-file format indications.
- **atomname_list:** String names can not exceed FOUR (4) characters per atom name, as per the *pdb*-file format indications.

Small overview of the *xyz_pdb* module functionality

```
$ Ducque --xyz_pdb INPUTFILE
The inputfile is read, the xyz file used as an input to output a well formatted pdb.
      ' * ' : mandatory
      ' + ' : additional in the GUI

INPUTFILE : [
  --xyz XYZ *
  The name of the file of the molecule you want to convert to pdb
      Example: --xyz dna_2endo.xyz

  --residue RESIDUE *
  The identifier of the nucleotide.
  (in this example, dXA is equal to deoxy-Xylose nucleic acid with an adenine base)
      Example: --atomID dA
      Example: --atomID dXA

  --atomname_list ATOMNAME_LIST *
  The ordered list of atoms that belong in the 'AtomName' column in a pdb file.
  The order follows the order of the atoms from the xyz file.
  NB : the responsability is with the end-user to see everything is correct.
      Example: --atomname_list O5', C5', H5'1, H5'2, C4' ··· , O3'
]
```

2.7 User implementation of custum nucleic acids

Adding a new chemistry to the repository

```
$ cd path/to/program/Ducque/src/builder/ # Go to the src files
$ vim builder_library.py   # Open builder_library.py in your preferred text editor
```

Adding a combination of nucleoside and linker moieties

```
# GO TO +- line 18
# ------------------------------------------------------------------------
#                   NUCLEOSIDE              ,             LINKER
# ------------------------------------------------------------------------
DH = path/to/Ducque # DUCQUE_HOME
codex_acidum_nucleicum = {
"dA" : [ DH + "dna_adenosine_2endo.json", DH + "dna_phosphate.json"],
"dG" : [ DH + "dna_guanosine_2endo.json", DH + "dna_phosphate.json"],
                              ...                              ...
# KEY : "foo" CHEMISTRY ABBREVIATION                  ; String
# VAL : [ "nuc", "link"] FOO.JSON ; List[String, String],
"foo" : [ DH + "foobar_nucleoside.json", DH + "foobar_linker.json"],
                              ...                              ...
}
```

Adding one ore more conformations of a nucleoside chemistry

```
# GO TO +- line 75
# ----------------------------------------
#       COMPLEMENTARY REPOSITORY
# ----------------------------------------
conformations_codex = {
"dA": [ DH + "dna_adenosine_2endo.json", DH + "dna_adenosine_3endo.json"],
"rG": [ DH + "rna_guanosine_3endo.json"],
                              ...                              ...
"foo": [ DH + "foobar_nucleoside.json"],
}
```

Adding the backbone atoms by which Ducque needs to build

```
# GO TO +- line 130
# ----------------------------------------
#       BACBKBONE REPOSITORY
# ----------------------------------------
backbone_codex = {
"DNA" : ["O3'", "C3'", "C4'", "C5'", "O5'"],
                              ...
"FOO" : ["V", "W", "X", "Y", "Z"],
}
```

3 Installing and running Ducque

3.1 Installation

Use `git` to clone the repository on your local machine. This can be done anywhere on the system, but is standardly done in the `$HOME/` directory. The set-up script is there to modify the runfile for Ducque. No sudo priviledges are required to run Ducque.
After all is set and done, either source the `~/.bashrc` or open a new terminal window and run the Ducque from the terminal.

```
$ git clone https://www.github.com/jrihon/Ducque.git # clone repo locally
$ cd Ducque              # Go to the Ducque directory, where you installed it
$ chmod a+x setup.sh     # Give permission to run as an executable script
$ ./setup.sh             # Run the setup.sh script
```

In your `~/.bashrc` .

```
export PATH=$PATH:path/to/program/Ducque/bin # in the $HOME/.bashrc
```

3.2 Environment

```
# Optional choice to keep envs separated
$ python3 -m venv Ducque            # virtual env using pip
$ conda create --name Ducque        # virtual env using conda

# Using the pip package manager
$ pip install numpy scipy

# Using the conda package manager
$ conda install -c numpy scipy

# Check if you have tkinter installed :
$ python3 -m tkinter
# Run the following command if the previous one was not succesful :
$ sudo apt-get install python3-tk
```

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.