**Python wrapper for NUPACK**

**user manual**

**Y. A. Rodenburg, Z. C. Félix Garza and H. W. H. van Roekel**

**Eindhoven University of Technology**

This user manual describes the use of the NUPACK 3.0 wrapper and the sequence generator for designing target sequences with complementary structure and target free energy. This wrapper is part of the Automated DNA Design Software (ADDS). ADDS is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. ADDS is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details (<http://www.gnu.org/licenses/>). NUPACK can be obtained from <http://www.nupack.org>.

**1 In advance**

**1.1 Installing Python**

These programs are made on a virtual machine running Ubuntu 10.04 LTS - the Lucid Lynx. Written in Python 2.6.5. The programs make use of the following libraries. They are essential to run the program.

• random – to generate pseudorandom numbers

• math – for complex mathematical calculation

• copy – for making deep copies

• time – time library

• os – for working with directories and files

• sys – for system operations

• Tkinter – for GUI interfaces

• Bio – BioPython, for bio-informatic tools.

• subprocess – for running subprocesses (NUPACK3.0) along the python process.

• pp – for executing parallel processes

Python can be installed by entering the following command in the linux terminal:

sudo apt-get install python2.6

Now all of the packages above should be included, except for BioPython. This can be installed by entering the following command:

sudo apt-get install python2.6-biopython

If Tkinter is not included in the default python installation, this can be installed by entering:

sudo apt-get install python2.6-tk

If pp is not included in the default python installation, this can be installed by entering:

sudo apt-get install python2.6-pp

To install BioPython locally in the ICMS cluster machine it is necessary to follow the next steps:

1. Download Biopython: [[[1]](http://biopython.org/wiki/Download)]

2. Move the biopython-1.59.tar.gz file into your home directory on the cluster machine, henceforth referred to as $.

3. Unpack the tar.gz file with the command:

$> tar -xvpf biopython-1.59.tar.gz

4. Move into the folder with:

$> cd biopython-1.59



5. Execute the command:

$/biopython-1.59> python setup.py install --prefix=$HOME

Now a local Biopython is up and running. However, it will not work because Tkinter does not work, and that is a result of the fact that xml does not work. In the folder that was created during installation ($/lib64/python2.5/site-packages), copy the xml folder from your own python 2.5 or up installation. This xml folder resides in <your-local-python-directory>/lib. The final thing to keep in mind is that every program that utilizes Biopython needs to have two extra lines of code in the header:

import sys

sys.path.append('$/lib64/python2.5/site-packages')

The same process should be followed to install pp locally in the ICMS cluster machine. The detailed steps can be found below:

1. Download Parallel Python: [http://www.parallelpython.com/content/view/18/32/]

**2.** Move the pp-1.6.4.tar.gz file into your home directory on the cluster machine, henceforth referred to as $.

**3.** Unpack the tar.gz file with the command:

$> tar -xvpf pp-1.6.4.tar.gz

**4.** Move into the folder with:

$> cd pp-1.6.4

**5.** Execute the command:

$/pp-1.6.4> python setup.py install --prefix=$HOME

Every file that uses pp should execute the following lines before using pp.

import sys

sys.path.append('$/lib64/python2.5/site-packages')

* 1. **Installing NUPACK 3.0**

Extract NUPACK3.0.tar in the desired folder. When in this folder in the linux terminal, type:

make

If you want to make use of NUPACK 3.0 exclusively, and not the Python wrapper, also set the path variable for NUPACK:

Export NUPACKHOME = /…/nupack3.0

1. **Python wrapper** 
   1. **NUPACK.py**

This class generates an object which is a wrapper for the NUPACK software suite. This object has the functionality to run the NUPACK binary *complexes*. It can also read the complexes output files and return them in a proper and understandable way.

**Due to a bug in NUPACK3.0, any file paths and directory names given to the programs must NOT contain spaces in their names.**

* + 1. **Parameter file**

The NUPACK object makes use of a special textfile (.txt) called *parameters.txt*. Here all the NUPACK parameters are defined for the use of complexes. The file *parameters.txt* can be given to the NUPACK class as a file string. If it is not given as an argument, the program prompts a file chooser to select the parameter file. Every parameter is on a newline. After the parameter, the value is written separated by " = " (space-equals-space). Parameter descriptions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | **Parameter** | | |  | | --- | | **Value(s)** | | **Description** |
| home | NUPACK home directory | The root of the NUPACK software suite directories.  e.g. “/home/user/Documents/NUPACK3.0/” |
| |  | | --- | | T | | |  | | --- | | Numeric ( > 0 ) | | Temperature in °C. |
| material | dna / rna / rna1999 | Material to use:  • rna (dna1995) – allows rna calculations at different temperatures.  • dna (dna1998) – allows dna calculations at different temperatures.  • rna1999 – used for rna calculations at 37°C and includes pseudoknot parameters. |
| ordered | True / False | Enable / disable the use of ordered complexes (possibility of crossing hydrogen bonds). |
| pairs | True / False | If this is enabled, base pair energies are calculated. |
| mfe | True / False | Enable / disable calculating the minimum free energy of each complex. |
| degenerate | True / False | True: calculate only the minimum free energy  False: calculate multiple low free energies. |
| |  |  | | --- | --- | | dangles |  | | none / some / all | If this is True, the program does only a time estimate. |
| timeonly | True / False | If this is True, the program does only a time estimate. |
| quiet | True / False | Enable / disable the screen output from NUPACK itself. |
| magnesium | Numeric | Mg++ concentration. |
| sodium | Numeric | Na+ concentration. |
| cutoff | Numeric | Only ensemble pair fractions at or above the cutoff value are saved in the output file (default is 0.001). |

The parameters can be in any order. Comment lines should start with “#”. Not every parameter has to be given, although it is better to do so. A parameter file should have the following format:

# parameters for nupack.py.

# Comment lines should start with '#'.

home = /.../wrapper/nupack3.0/

T = 38.5

material = dna

ordered = True

pairs = False

mfe = True

degenerate = False

dangles = all

timeonly = False

quiet = True

sodium = 0.05

magnesium = 0.007

* + 1. **Running the program**

The NUPACK object can be initialized in silent mode or not, depending on the use. Silent mode simply means the program does not return screen output. If the program is looped via another program, it is better to set the NUPACK object to silent mode, as screen output slows the program down. Here is prefix the directory of the NUPACK input files (defined in the NUPACK 3.0 user guide) without the file extension (e.g. /wrapper/model/sequences).

The NUPACK object is initialized in the following way, where "..." are the NUPACK *complexes* parameters defined in section 2.1.1. These parameters can be used as default if this class is imported in other algorithms.

import randomizer

nupackObject = randomizer.NUPACK(prefix, paramFile=None, silent=False, ...)

In this case the file *NUPACK.py* should be in the same folder as the current work directory.

* + 1. **Functions & output**

If the NUPACK instance is called, the NUPACK instance allows the user to run the following functions:

* NUPACKObject.runComplexes()
* NUPACKObject.readLastOutput()
* NUPACKObject.setSilent(value)

runComplexes() runs the complexes binary from the NUPACK 3.0 software suite. It uses the given file prefix to read the in-file (.in) and eventually the list-file (.list) (described in the NUPACK 3.0 user guide). readLastOutput() reads and returns the indexes of the used strands for the permutations by NUPACK, the secondary structures and the free energy of those. If the NUPACK object is not set silent, the readLastOutput() function returns a screen output that looks like the following:

Strand 1: Strand 2: Free Energy: Structure:

sequence 1 -1.21855746e+00 ........(((.......))).

sequence 2 -7.03881171e-02 ...........

sequence 1 sequence 2 -1.47595373e+01 ...........((((((((((.+.))))))))))

The return value is a two-dimensional list, with the different permutations on the first level, shown as rows in the screen output above. The second level contains the columns, with the first strand on index 0, the second strand on index 1, and on index 2 (or 1, depending on the use of multiple strands for a permutation) the free energy, index 3 (or 2) contains the predicted secondary structure calculated by NUPACK complexes. This is the standard output. The pure NUPACK output is stored in the current work directory in a collection of files (described in the NUPACK user guide).

Whether the program runs silent or not, this can be changed by calling the function setSilent(value). The value given as argument is a Boolean.

* 1. **seqGen.py**

seqGen.py contains a class called “Main”. This is created to generate and classify sequence sets by a simulated annealing algorithm and to loop these sequences a given times through the NUPACK complexes program, using the NUPACK python class specified before. Parallel to the analysis executed by NUPACK, a critons assessment is performed by a critons assessment algorithm that makes use of the critonsCheck python class described in section 2.3 of this manual.

* + 1. **Input**

The class is executed by simply calling the class *Main* with a starting temperature for the simulated annealing, a number of NUPACK runs (and so a number of sequence generations), a starting temperature, an end temperature, a factor value for the critons evaluation, a maximum complex size (1 or 2), a sequence file string, a binding file string, a list of runs at which the program makes a distinct save, and a mode parameter which defines the mode the program runs in. This gives the user the choice of running the default simulated annealing trial (mode="SA") or a random sequence trial (mode="random"), which will run the *randomizer.py* program. The last five parameters are optional. If there are no files specified in the class arguments, the program will open a file chooser. These files specify the sequences to be generated, and the properties of a target duplex.

seqGen = Main(nRuns, T\_start, T\_end, lambdaFactor, maxCxSize=1, seqFile=None, bindingFile=None, saveRuns=[ ], mode=”SA”)

The sequence file is a text file which must have a specific format and contents. It contains three tabbed columns with the name of the sequence in the first column. The second column includes the sequences with undefined bases as an “N”, and bases which are already specified and thus cannot be changed noted as an A, T, G or C. The third column contains the strand type, which is a **T** for **template**, a **B** for a **binding** strand or an **I** for **inhibitor**. **All sequences should be noted from 5’ to 3’**.

Seq\_A NNNNGACTCNNNNNNGACTCNNNN T

Seq\_B NNNNGAGTCNNNN B

Seq\_C NNGAGTCNNNN I

Seq\_D NNNNNNNNNNNNNNNNNNNNNNNN T

The binding file, also a text file, contains the permutations that indicate in which combinations the sequences should form duplexes with each other. Four tabbed columns where the first two columns contain the names of the sequences defined in the sequence file. These names should only be next to each other if they are supposed to form a duplex. **The template strand should always be the most left.** The third column comprises the target secondary structure, given by a dot-parenthesis-plus notation. This notation implements the target binding bases of 2 strands. The program needs this to identify which bases are supposed to be complementary to each other. The fourth column has the target free energy of the duplex. A fifth column is required if the duplex contains an inhibitor strand. This defines the binding type, which is an "**I"** for **inhibiting**, and a "**B"** for **binding**. This information is needed by the program because inhibitors overlap other strands. So this information makes sure the program adjusts the sequences in the right order after a mutation. The next text square shows an example of a binding file.

Seq\_A Seq\_B ...........(((((((((((((+)))))))))))))

Seq\_A Seq\_C (((((((((((.............+))))))))))) B

Seq\_D Seq\_C .....(((((((((((........+))))))))))) I

* + 1. **Output**

After running the program, multiple output files are generated:

* Multiple NUPACK output files, depending on the given parameters, called **cxFile.in -.list, -.cx, -.ocx, -.ocx-mfe, -.ocx-key**. The explanation of these files can be found in the NUPACK manual.
* Multiple NUPACK output files containing distinct saves defined by the saveRuns parameter, called **runXX.in -.cx, -.ocx-mfe**, where XX is the run index.
* Three output files called **LowestError.in -.cx, -.ocx-mfe**. These files contain the case that resulted in the minimum error.
* **GC\_Content.txt** containing for each strand a column with the GC content stored for each run.
* **Free\_Energies.txt** containing for each strand a column with the free energy stored for each run.
* **Errors.txt** containing two columns, the first column comprises the total error per run, and the second column has the delta error.
  1. **critonsCheck.py**

This class is responsible for the appraisal of all the critons derived from each template strand. The main function (*checkCritons)* of this class comprises a set of functions that allow the identification of the duplicated critons and the evaluation of the template strands in terms of the number of duplicated critons that are of a repeated base type, i.e “AAA”, or “TTT”, or “GGG”, or “CCC”, and those that are not. Within the main function a cost function related to the critons is computed. The value calculated from the cost function is then set as the output value of the *checkCritons* function.

An object of this class is initialized from the seqGen.py in order to access the *checkCritons* function and execute the critons assessment and compute of the error value related to the critons.

**2.3.1 Input**

The *CritonsCheck* object is initialized from the class *Main* of seqGen.py by calling the *CritonsCheck*  class with the current iteration for sequence generation, the value of the last iteration that will be performed as an indication of when to save the information yield by the *checkCritons* function, a factor value (*lambdaFactor*) for the computation of the critons related cost function, the file names for the data and information files where the critons assessment data will be saved, a critons size *lc* , a strands object containing the sequences obtained by the simulated annealing algorithm, and the three set of sequences that correspond to the nickase recognition sequences and further overlapping between template strands. The last five parameters are optional. If there is no value defined for the critons size, this parameter will have a default value of 3. Similarly for the strands object, if no set of sequences is defined for this object, it will be set as empty and the sequences will then be obtained from a default file.

critonsCheckObject = critonsCheck(runNum, nRuns, lambdaFactor, dataFileName, infoFileName, lc = 3, strandsObject = strands, excTemp1=['GACTC'], excTemp2=['CTCGA','GACT','GACTC'], excTemp3=['CTCGA','GACTC'])

**2.3.2 Output**

* The main function (*checkCritons)* of the class returns the variable **critonWeight**, comprising the value that will be added to the global error function in the seqGen.py class.
* **CritonsDataFile.txt** containing the raw data related to the number of critons for the template strands in the system, as well as their number of repeated critons, their number of repeated base critons and duplicated base adjacent critons, the number of duplicated critons between strands, and the contribution of the critons to the global cost function. The file is made in the first and last iterations of the program, as well as every 1000 iterations.
* **CritonsInfoFile.txt** includesa summary of the results obtained by the critons assessment for each of the template strands, as well the contribution of the critons in each template to the global error function. The file is made in the first and last iterations of the program, as well as every 1000 iterations.