Unversitat Politècnica de València

ETSIAMN

Grado en Biotecnología

Automated sequence design of nucleic acid hybridization reactions for microRNA detection

Biotechnology Bachelor's Thesis

(Trabajo de Fin de Grado en Biotecnología)

Academical year 2018 – 2019

STUDENT: Lucas Goiriz Beltrán

TUTOR: Prof. Javier Forment Millet (UPV)

SUPERVISOR: Dr. Guillermo Rodrigo Tárrega (CSIC)

Valencia, July 2019

Title: Automated sequence design of nucleic acid hybridization reactions for microRNA detection

Abstract (English):

microRNA (miRNA) can be found in a variety of biological samples and then they represent important molecular markers for early diagnostic strategies. This work *(TFG)* explores a novel approach based on nested non-enzymatic and enzymatic biochemical processes in vitro. In particular, an automated sequence design algorithm of nucleic acid hybridization reactions for microRNA detection is developed.

Abstract (Spanish):

Los microRNAs (miRNAs) pueden ser hallados en una gran variedad de muestras biológicas y suponen una fuente importante de marcadores moleculares para estrategias de diagnóstico tempranas. En este trabajo *(TFG)*, se explora un abordaje novedoso basado en procesos bioquímicos anidados enzimáticos y no enzimáticos in vitro. Particularmente, se desarrolla un algoritmo de diseño de secuencias automatizado para reacciones de hibridación de ácidos nucleicos para la detección de microRNA.

Key words (English): microRNA, nucleic acid circuit, algorithm, sequence design, Python.

Key words (Spanish): microRNA, circuito de ácidos nucleicos, algoritmo, diseño de secuencia, Python.

TFG Author: Student: Lucas Goiriz Beltrán.

Location and Date: Valencia, July 2019.

Academic Tutor: Prof. Javier Forment Millet (UPV).

Supervisor: Dr. Guillermo Rodrigo Tárrega (CSIC).

Aknowledgements

I would like to thank firstly Javier Forment for his implication in my passion towards bioinformatics and programming. The dedication during his lessons inspired to keep learning programming languages and being open minded towards new ideas and programming approaches.

Second, I would like to thank Guillermo Rodrigo for accepting me into his team although I was just an undergraduate who couldn't possibly be of much help. However, I felt welcome from the first moment and, although struggling sometimes, I tried my best to not to disappoint him. In addition, it is thanks to Guillermo that I discovered the world of systems biology, the complex world behind scientific publications and the life as a researcher. Guillermo is a role model I will always be looking up to.

Third, I would like to thank to my lab mates, especially Roser, who was always ready to help my in whatever issue I encountered and offered me a piece of chocolate whenever she saw me tired or uninspired. I would feel truly blessed if I am offered the opportunity to come back to this team, as it felt like a tiny family.

Fourth, I would like to thank my friend/"disciple"/roommate Enrique, who had to deal with my frustration whenever the algorithm didn't behave as expected. It was always of great help having a pair of eyes on the screen pointing out possible sources of errors while debugging.

Last, but not least, I would like to thank my friends for these amazing 4 years at the University. Especially Ramón and Arcadio, who were with me in the good and the bad moments: during the long study evenings and the long weekend nights. Please, never change.

Index

Index of Figures

Index of Tables

Index of Boxes

1 Introduction

DNA nanotechnology is a promising field in which DNA strands are employed with the aim of manipulating the temporal and spatial distribution of matter within a system, giving rise both to self-assembled nanometer-scale structures and autonomous reconfigurable devices. The main interest of the self-assembled structures is their stability at the equilibrium state (structural DNA nanotechnology). On the other hand, the main interest of the autonomous reconfigurable devices relies not on the equilibrium states but on the non-equilibrium states that allow the device to switch from one equilibrium state to another (dynamic DNA nanotechnology), while employing non-covalent interactions (Zhang & Seelig, 2011).

These dynamic DNA nanotechnology-based devices employ DNA hybridization, strand displacement and dissociation in order to switch between said equilibrium states (Zhang & Winfree, 2009) and their usage as nanoscale devices with the aim of controlling biological circuits in vivo, building nanoscale chemical circuits or analyzing biological samples is rising (Seelig *et al.*, 2006) due to the wide characterization of their Watson-Crick hybridization thermodynamics of base pairs and the predictability of single stranded DNA and double stranded DNA secondary structures, allowing thus a rational design of structures and interactions based on the primary nucleic acid sequence (Zhang *et al.*, 2007; Zhang & Seelig, 2011; Zhang & Winfree, 2009). Furthermore, the elaboration of these DNA devices is getting more feasible due to the exponential reduction of oligonucleotide synthesis and purification costs.

At the present time, DNA-based synthetic molecular circuits do not approach the complexity and reliability of modern electronics (Seelig *et al.*, 2006). However, they present a promising alternative as control devices in biological systems as they function both in vitro and in vivo, store signal information in molecule concentrations and conformations, and their complexity ranges from low component systems, where signal is produced only in the presence of the appropriate input, up to complex systems that include various logic gates that evaluate the presence of various possible inputs that trigger a wide combination of outputs, opening the door towards biological computing.

The most important reaction that allows the dynamic behavior that permits these circuits' performance is known as strand displacement, which is the process of hybridization of two strands with partial or total complementarity while displacing pre-hybridized strands, which can act as triggers for further strand displacement reactions (Zhang & Seelig, 2011).

This process usually initiates at short single stranded domains where interacting strands have total complementarity, known as toeholds, and progresses until reaching total strand hybridization. Therefore, it is a reaction that does not require enzymes as it exclusively depends on the biophysics of DNA and whose kinetics can be controlled by varying the sequence and length of toeholds (Zhang & Winfree, 2009; Zhang & Seelig, 2011; Srinivas *et al.*, 2013). An illustrated example is provided in ***Figure 1***.

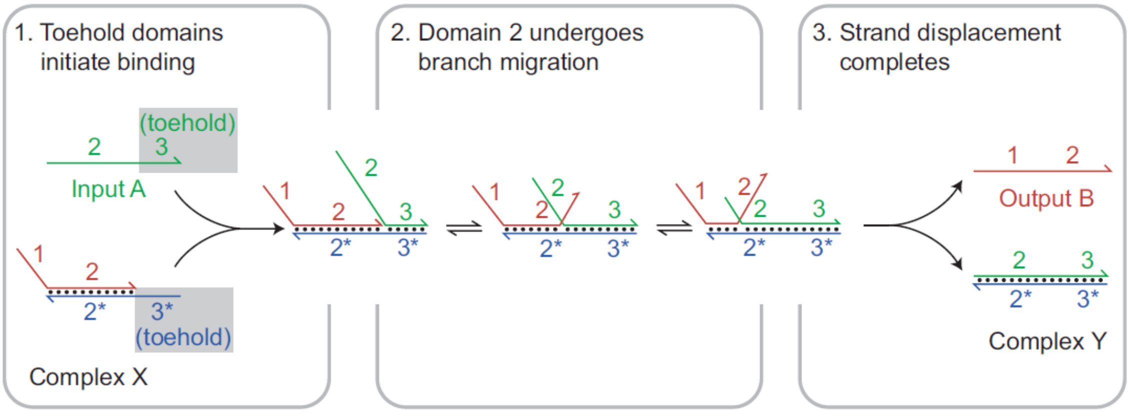

1. Toehold domains initiate binding
2. Domain 2 undergoes branch migration
3. Strand displacement completes
(toehold)
2
3
Input A
1
2
2*
3*
(toehold)
Complex X
1
2
3
2*
3*
1
2
2
3
2*
3*
1
2
2
3
2*
3*
1
2
Output B
2
3
2*
3*
Complex Y

Figure 1*: Strand displacement reaction example. Modified from Zhang & Seelig (2011)*

The driving forces of strand displacement reactions are the enthalpy gain by forming base pairs and the entropy gain by releasing pre-hybridized strands, meaning that the reaction is stable, up to a certain degree, to environmental changes such as salt concentration and temperature, which typically modify DNA hybridization strength (Zhang *et al.*, 2007; Zhang & Seelig, 2011).

Both driving forces are dependent on the presence of input, as it is the one strand that allows the formation of new base pairing and the liberation of pre-hybridized strands. Therefore, the reaction is limited by the amount of input present initially and when reaching equilibrium (which typically means the consumption of all the input as this provides a more stable thermodynamic state of the system), the reactions cease, and the circuit stops working.

However, if the application requires it, the input species may be replenished by mechanisms such as transcription, which unlike strand displacement reactions, consume a standardized energy source (ATP) with the disadvantage of needing the corresponding enzyme for this task (Zhang & Seelig, 2011).

Thanks to the strand displacement mechanism and its characteristics, DNA-based synthetic molecular circuits can be employed as systems of signaling cascades where a low concentration, and initially undetectable, input signal (usually a single DNA or RNA strand) may be amplified to, for example, a measurable fluorescence signal.

A particularly interesting application of these circuits is the early detection of biomarkers for early diagnosis like cancers and many neurodegenerative diseases, such as Alzheimer's, as there are serum miRNA biomarkers available, shown in ***Table 1***:

Table 1*: miRNA biomarkers for cancers and neurodegenerative diseases (Kumar et al., 2013; Qiu et al., 2015; and Wittman et al., 2010)*

Disease	**Potential serum miRNA biomarkers**
Colorectal cancer	miR-17-3p; miR-92; miR-29a; miR-92a
Diffuse large B-cell lymphoma (DLBCL)	miR-21; miR-155; miR-210
Lung cancer	miR-25; miR-223; miR-17-3p; miR-21
Breast Cancer	miR-155; miR-195
Prostate Cancer	miR-16; miR-34b; miR-92a; miR-92b
Amyotrophic Lateral Sclerosis (ALS)	miR-206; miR-155
Huntington's Disease	miR-9; miR-22; miR-128 HTT; miR-132
Parkinson's Disease	miR-133b; miR-107; miR-34; miR-205
Alzheimer's Disease	hsa-let-7d-5p; hsa-let-7g-5p

These types of diseases are in the top 10 death causes in western countries (Heron, 2018; Soriano *et al.*, 2018) and are often diagnosed much too late due to the late appearance of the symptoms, which in certain neurodegenerative diseases take up to 20 years to appear (Kumar *et al.*, 2013). Furthermore, the complexity of these diseases lowers the success rates of the available treatments as the response can greatly vary between individuals. ***Figure 2*** illustrates how neurodegenerative diseases and cancers dominated the causes of death in Spain in 2016.

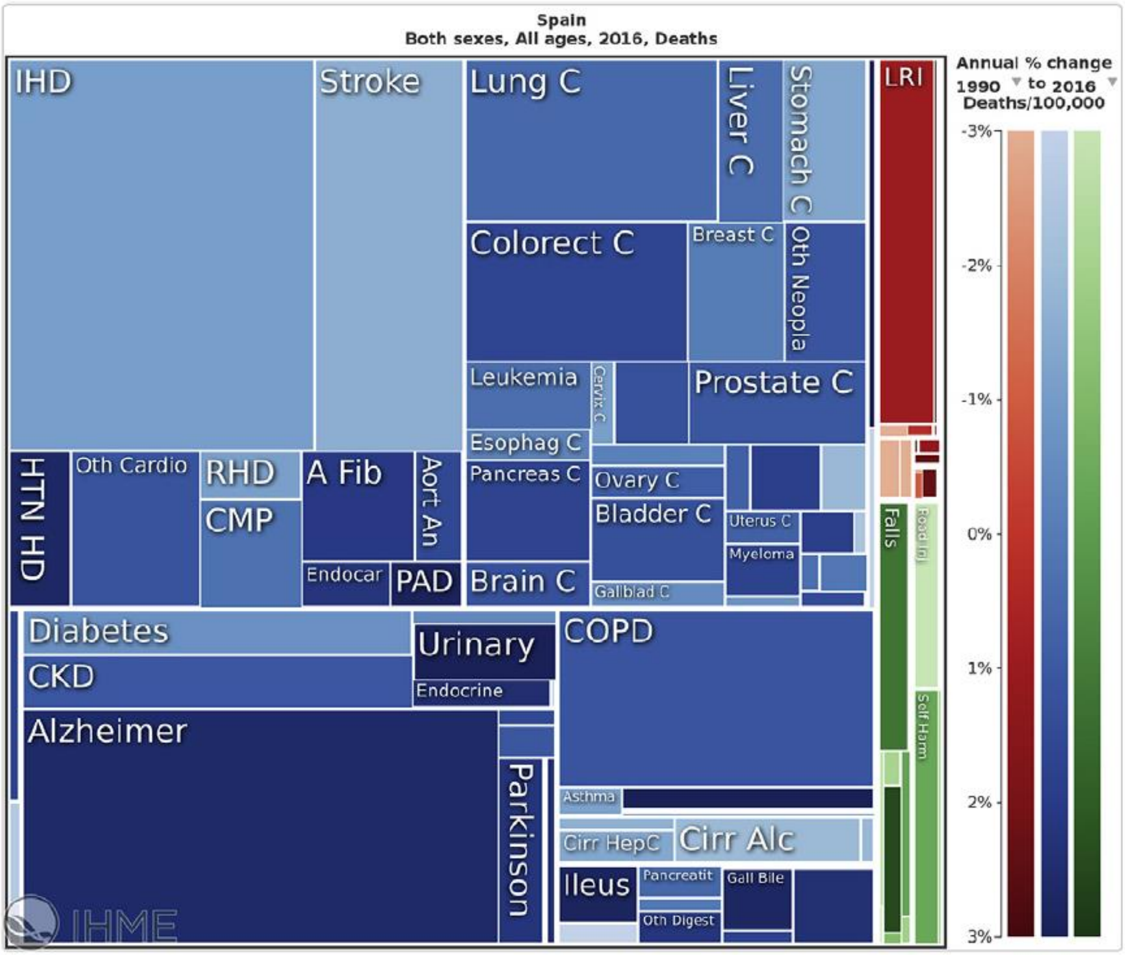


Figure 2*: Main causes of death in Spain 2016. Modified from Soriano et al. (2018)*

This fact creates a need for a personalized treatment, which in turn requires a precise diagnostic of the disease sub-category. Modern effective diagnostic strategies rely on the use of biomarkers such as proteins or on gene expression profiling by means of microarray technology. Nevertheless, these approaches are invasive and laborious as they require a tissue biopsy for their analysis while lacking the precision needed for a personalized treatment. It is through the disadvantages of those biomarkers that miRNAs are gaining popularity as a novel source of circulating biomarkers for diagnostics.

miRNAs are single stranded, non-coding regulatory RNA molecules of around 22 nucleotides in length. Their biogenesis in animals begins at the transcription of miRNA genes in the form of long primary miRNA (pri-miRNA) which are processed by the Microprocesor complex (consisting of RNase III Drosha and the double stranded RNA binding domain DGCR8) into pre-miRNA, which are short oligonucleotides of 70 nucleotides in length. Next, the pre-miRNAs are exported into the cytoplasm by means of exportin-5 (EXP-5) where they are further processed by RNase III Dicer into mature miRNA (Catalanotto *et al.*, 2016; Wahid *et al,*. 2010). miRNA biogenesis in animals is illustrated in ***Figure 3***:

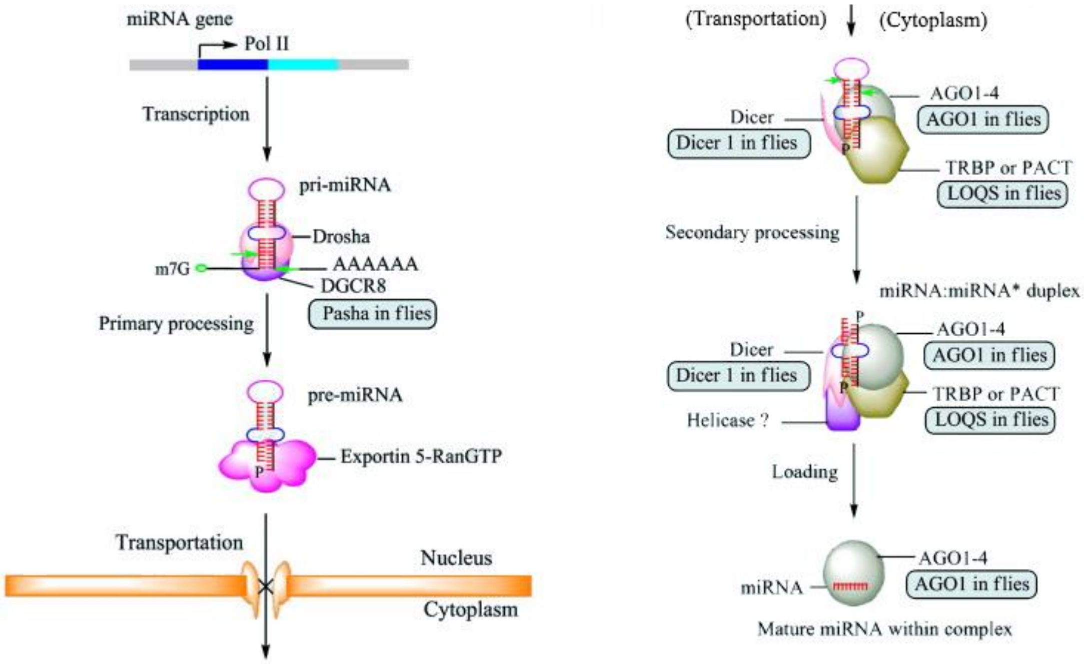


Figure 3*: Steps of miRNA biogenesis in animals. Modified from Wahid et al. (2010)*

miRNAs can be expressed either ubiquitously or in a tissue/cell specific manner while also displaying various expression patterns along tissues/cells which can also vary with time (Pockar *et al.*, 2019). They act as gene expression modifiers at post-transcriptional level either by binding to the 3' UTRs of the mRNA they target (Kumar *et al.*, 2013; Wahid *et al,*. 2010; Wang et al., 2012) or by recruiting mRNA silencing complexes such as the RISC complex (Catalanotto *et al.*, 2016; Wittmann & Jäck, 2010). Around 4% of genes present in the human genome encode miRNAs, and a single miRNA can be involved in the regulation of up to 200 mRNAs (Kumar *et al.*, 2013).

This fact is due to the stability of miRNA in biological fluids, either caused by RNA binding proteins (such as NPM1, HDL or Argonaute2), transporter microparticles or exosomes (small membraned vesicles) (Wittmann & Jäck, 2010), which allows them to be exported out of the cell and affect mRNA expression of distant cells. miRNAs play a regulatory role in many biological processes being therefore highly conserved during evolution, although it is believed that mechanisms through which they function are different (Pockar *el al.*, 2019). Furthermore, neurodegenerative diseases and cancers have an altered miRNA profile when comparing with adjacent healthy tissue. All these characteristics plus the simplicity of extracting a blood sample from a patient, make miRNA a very attractive source of biomarkers to consider for early diagnostics.

With the rising of the usage of miRNA as biomarkers, several tools and technologies are advancing towards a more precise quantification of miRNA. Traditional laboratory methods are qPCR, microarray technology and NGS. These technologies require as a first step an amplification of the miRNA by means of RT-PCR into cDNA.

In qPCR technology, the sample undergoes a consecutive amplification while measurements in real time are taken by means of fluorescent probes. The main disadvantage of this technique is that the short length of miRNAs conditions the primer design, which must not form primer dimers. In addition, they must ensure a low detection threshold (Balcells *et al.*, 2011; Chen *et al.*, 2005; Redshaw *et al.*, 2013).

Microarray technology depends on a hybridization reaction of the sample with DNA probes anchored to a solid surface. The main disadvantages are involved with the need of specialized equipment, the different probes available, the lack of hybridization procedure standardizing and the challenge of data normalization due to the weak expression levels and low concentration of miRNA (Draghici *et al.*, 2006; Wang & Xi, 2013; Wu *et al.*, 2013).

NGS technology for miRNA quantification consists in sequencing the miRNA found on a sample. It is becoming the preferred method as costs are being greatly reduced. Nevertheless, there are great disadvantages, which involve the NGS data analysis and its lack of standardization (Chatterjee *et al.*, 2015; Li *et al.*, 2015).

The technologies aforementioned are difficult to implement in the clinic, at home or in underdeveloped countries, as they require specialized apparatus and staff, which hinders the quick obtention of results. The need of a more simplistic and quick manner of detecting miRNA and amplifying the signal without a prior polymerase mediated amplification led to the exploration of a non-enzymatic miRNA detection based on strand displacement synthetic DNA circuits.

However, these circuits present major issues when tested in vitro as hybridization may not be perfectly specific and undesired hybridizations may happen. In addition, oligonucleotide synthesis errors, such as deletions, deaminations or depurinations strongly affect the performance as the circuit strongly depends on its components' base sequence (Zhang *et al.*, 2007; Zhang & Seelig, 2011).

Another disadvantage is that the sequence of a circuit's components is strongly dependent on the sequence provided by the input(s), meaning that certain circuits may underperform or "leak" signal by spontaneous fluctuations caused by the low robusticity of their base sequence (Seelig *et al.*, 2006; Song *et al.*, 2018; Wang *et al.*, 2018).

All these issues plus the importance of an appropriate sequence design forces a strong *in silico* approach of every system design prior to any *in vitro* testing. There is a need of an automated sequence design algorithm based on *in silico* simulations of the proposed system to ease future fine tuning based on experimental measures.

1.1 Python as a Bioinformatics tool

For the resolution of the aforementioned biological problem, the present work employs the Python programming language. Python has several characteristics that makes it more suitable as a programming language in bioinformatics than other languages like JAVA or C. First, its comfortable readability (allowing a better understanding of the code and improvement by scientific peers); second, it is open source (which makes it available to any user); third, it is cross platform (allowing Python programs to run in any kind of machine as long as they have the Python interpreter) and fourth, it has a growing scientific community (Bassi, 2010; Ekmekci *et al.*, 2016), which creates modules and libraries, such as BioPython or SciPy, with the purpose of being employed in bioinformatics and many other fields.

However, since Python is an interpreted programming language, it has the drawback of having a lower performance than compiled languages (such as C), which translates into longer execution time for the same results. Nevertheless, for small programs in modern machines this difference is not really significant as Python may take up around 10 seconds to finish while C only takes up 1 (Bassi, 2010). If the code development time is taken into account, Python results to be much faster due to the simplicity in code development.

2 Objective

To generate an algorithm that automatically designs a DNA circuit for miRNA detection based on a sequence input while avoiding signal leak and being overall robust.

3 Materials and Methods

3.1 Computational Resources

3.1.1 Hardware

For the elaboration of the algorithm, the following platforms were employed:

A) Computer with Ubuntu 16.04.6 LTS Operative System with 23Gb RAM and Intel® Xeon® E5504 processor.
B) Laptop with Windows 10 Operative System with 8Gb RAM and Intel® Core™ i5 7200U processor.

3.1.2 Software

The Python version employed in this work was 3.5.2 (PYTHON SOFTWARE FOUNDATION, 2019) altogether with the following libraries:

- The Python Standard Library (PYTHON SOFTWARE FOUNDATION, 2019), where the following modules were used: sys, subprocess, random, datetime, time and math.
- ViennaRNA 2.4.10 Python3 Library (Lorenz *et al*., 2011).
- Potly Python Open Source Graphing Library (PLOTLY, 2019).
- SciPy Fundamental Library for Scientific Computing (SCIPY, 2019).

Additional software employed in this work includes NUPACK 3.2.2 (Dirks & Pierce, 2003; Dirks & Pierce, 2004; Dirks *et al.*, 2007) with a code wrapper for its implementation in Python courtesy of Salis *et al.* (2009).

3.2 Simple Circuit Components

The initial state of the system consists in an equilibrium in which the complexes sensor-transducer and clamp-T7p are stable. The addition of miRNA to the system causes a disruption of these complexes by means of strand displacement reactions that occur due to the lower minimum free energy (MFE) of the possible complexes to be formed in the presence of the input, forming thus different complexes until reaching a new equilibrium state. The output strand, T7p, will then act as a primer sequence for a DNA template which will be transcribed with the objective of carrying out a signal amplification.

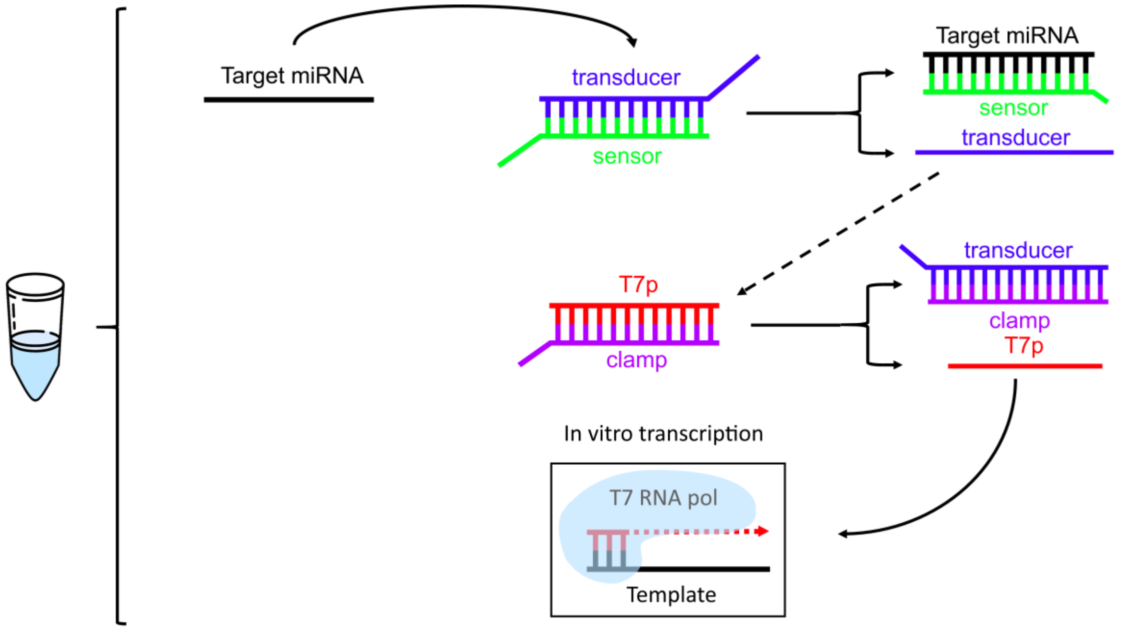


Figure 4*: Simple miRNA detection circuit components and interactions*

3.3 Simple Circuit Initial Sequence Design

The initial sequences for the components of the circuit originate from a “master sequence,” which is in turn formed by the joining of the target miRNA sequence and the T7 Phage Promoter sequence (T7p). Sensor and clamp sequences originate as the reverse complementary sequences of sections from the “master sequence”, while transducer is merely a section of the master sequence. An example is shown below:

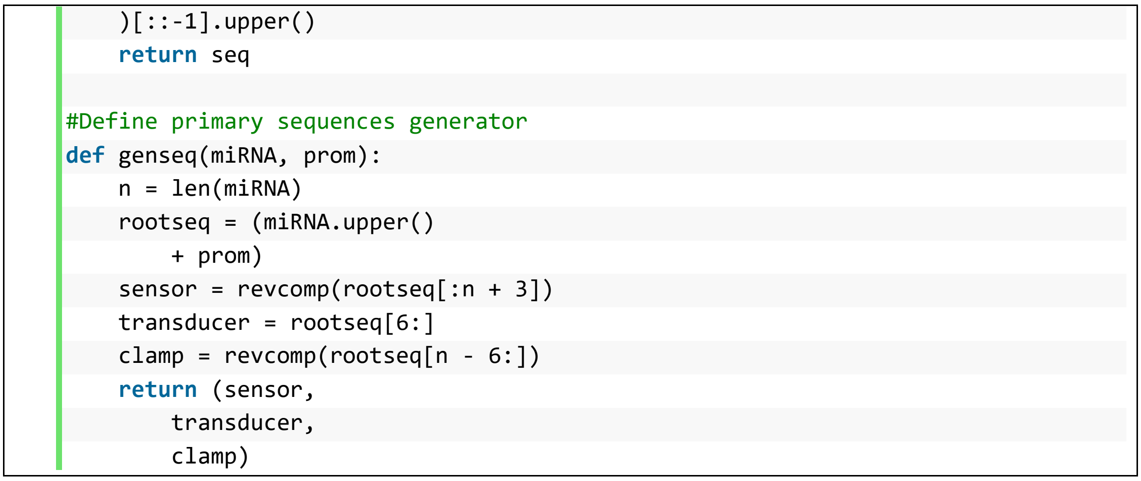
Table 2: *Circuit components initial sequence generation example*

```
Master:                 TGGAGTGTGACAATGGTGTTTGGCGCTAATACGACTCACTATAGG
miRNA (5’-3’):          TGGAGTGTGACAATGGTGTTTG
sensor (3’-5’):         ACCTCACACTGTTACCACAAACCGC
transducer (5’-3’):           GTGACAATGGTGTTTGGCGCTAATACGACTCACTATAGG
clamp (3’-5’):                          ACAAACCGCGATTATGCTGAGTGATATCC
T7p (5’-3’):                                  GCGCTAATACGACTCACTATAGG
```

Note that the nucleotides highlighted in yellow are the ones that will serve as toeholds for strand displacement reaction initiation. The first toehold (marked at sensor) will promote miRNA adhesion and displacement of transducer. The second toehold (marked at clamp) will promote transducer adhesion (only if this strand is completely free, as the complementary sequence of the toehold is hidden when transducer is part of the complex sensor-transducer) and T7p displacement. The code needed in order to perform this task is shown in ***Box 1***:

```
#Define reverse complementary generator
def revcomp(seq):
    seq = seq.upper(
    ).replace('A','t'
    ).replace('T','a'
    ).replace('G','c'
    ).replace('C','g'
```

This box continues on the next page



```
    )[::-1].upper()
    return seq

#Define primary sequences generator
def genseq(miRNA, prom):
    n = len(miRNA)
    rootseq = (miRNA.upper()
        + prom)
    sensor = revcomp(rootseq[:n + 3])
    transducer = rootseq[6:]
    clamp = revcomp(rootseq[n - 6:])
    return (sensor,
        transducer,
        clamp)
```

Box 1*: Python functions for initial sequence generation*

3.4 Mathematical Approach and Objective Function

The circuit itself is evaluated in its equilibrium state and depending on the presence of input (miRNA) or not. Thus, the equilibrium states for an ideal working circuit are illustrated in ***Figure 5***.

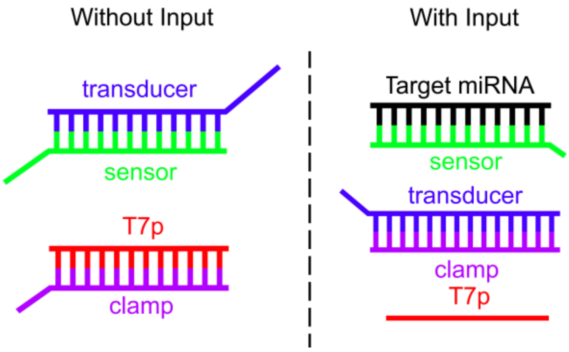


Figure 5*: Ideal equilibrium states for circuit components*

A Good way to evaluate the capability of the system to shift between both equilibriums by means of the addition of the target miRNA is to calculate the probabilities of the formation of the complexes found in the ideal case equilibrium in which the miRNA is present, against the complexes formed in absence of the miRNA. This calculus is done by means of a ratio between the Boltzmann function of the complex of interest and the Boltzmann functions of all other possible complexes involving each of the strands participating in the complex of interest. In addition, this ratio can be simplified as most possible complexes aren't spontaneous and thus, their Boltzmann values are negligible. For miRNA-sensor and transducer-clamp complexes, the probabilities of complex formation are the following:

$$P_1 = \frac{e^{-\beta\Delta G_{miRNA-sensor}}}{e^{-\beta\Delta G_{miRNA-sensor}} + e^{-\beta\Delta G_{sensor-transducer}}} \qquad \text{Eq. (1)}$$

$$P_2 = \frac{e^{-\beta\Delta G_{transducer-clamp}}}{e^{-\beta\Delta G_{transducer-clamp}} + e^{-\beta\Delta G_{clamp-T7p}}} \qquad \text{Eq. (2)}$$

Where:

β: the inverse of the product between temperature (K) and Boltzmann constant (k_B) ≈ 1,69

ΔG_{i-j}: MFE value of complex i-j.

An increment in the probabilities is to be achieved by means of increasing the MFE of the complexes present after the addition of input miRNA. To avoid an increment due to the reduction of the MFE of the complexes present prior to the addition of input miRNA, a set of “artificial probabilities” are calculated, which are based on a simulated MFE that acts as a minimum requirement, forcing therefore the complex MFE to be close to the simulated value. In the case that the Boltzmann function value of the complex was higher than the simulated one, the probability would be equal to 1:

$$P_3 = min\left(\frac{e^{-\beta \Delta G_{sensor-transducer}}}{e^{-\beta L_1 \Delta G_{bp}}}, 1\right) \qquad \text{Eq. (3)}$$

$$P_4 = min\left(\frac{e^{-\beta \Delta G_{clamp-T7p}}}{e^{-\beta L_2 \Delta \mathrm{G_{bp}}}}, 1\right) \qquad \text{Eq. (4)}$$

Where:

ΔG_{bp}: the average MFE of each base pair in a structure ≈ -1,25

L_1: length of the maximum possible interaction zone in sensor-transducer.

L_2: length of the maximum possible interaction zone in clamp-T7p (equivalent to the length of T7p).

Additionally, to avoid spontaneous transducer-clamp complex formation (the main source of signal leakage in this construction) promoted by the liberation of the toehold binding site hidden in the sensor-transducer complex structure, the dot and bracket structure of the sensor-transducer complex is evaluated. The number of unpaired nucleotides of a total of 6 in the toehold zone of transducer are counted:

$$T_{(struct(sensor-transducer))} = \sum_{i=6} "." \qquad \text{Eq. (5)}$$

Where:

“.” : represents the unpaired nucleotides in dot and bracket structure.

The Objective Function to optimize employs all 5 terms and is defined as:

$$F_{score} = P_1 P_2 P_3 P_4 \left(\frac{6-T}{T}\right) \qquad \text{Eq. (6)}$$

The implementation of the Objective Function in code is shown in ***Box 2***:

```
#Vienna parameters:
    #Mathews parameterfile
RNA.read_parameter_file(
    '~/ViennaRNA/misc/dna_mathews2004.par')  #Substitute '~' by your directory
    #No dangles
RNA.cvar.dangles = 0
    #No coversion from DNA into RNA
RNA.cvar.nc_fact = 1

#Global variables employed throughout the code
#Define circuit sequence names
guide = ['miRNA','sensor','transducer','clamp','T7p']
```

This box continues on the next page

```
#Boltzmann function parameters
BETA = 1/0.593
Num_e = 2.7182818284590452353
DGbp = -1.25

#Define Boltzmann function
def bolfunc(seq1, seq2, seq_DG):  #seq_DG is a dictionary containing the MFEs
    Pairkey = (seq1
        + '_'
        + seq2)

    Numerator = Num_e**(-BETA*seq_DG[Pairkey])
    Denominator = Numerator

    if seq1 == guide[0]:
        SecondKey = 'sensor_transducer'
        Denominator += Num_e**(-BETA*seq_DG[SecondKey])

    if seq1 == 'transducer':
        SecondKey = 'clamp_T7p'
        Denominator += Num_e**(-BETA*seq_DG[SecondKey])

    func = Numerator/Denominator
    return func

#Define function for probability calculation for secondary pairments
def probfunc(seq1, seq2, seq_DG, seqs): #seqs contains the sequences
    Pairkey = (seq1
        + '_'
        + seq2)
    Numerator = Num_e**(-BETA*seq_DG[Pairkey])

    if seq1 == 'sensor':
        L = 19
        Denominator = Num_e**(-BETA*L*DGbp)

    if seq1 == 'clamp':
        L = len(seqs[guide[4]])
        Denominator = Num_e**(-BETA*L*DGbp)

    func = Numerator/Denominator

    if func > 1:
        func = 1

    return func

#Define toehold score
def toeholdscore(name, seq_ss):  #seq_ss contains the complex’ structures
    DIST = (len(seqs_preit['transducer'])
        - len(seqs_preit['T7p']))
    struct = seq_ss[name].split('&')[1][(DIST-6):DIST]

    j = 0

```

This box continues on the next page

```
    for symbol in struct:
        if symbol == '.':
            j += 1
    return j

#Define Packing and Scoring function
def scorefunc(seqs):
    seq_DG = {}  #Dictionary where the MFEs will be stored
    seq_ss = {}  #Dictionary where the structures will be stored
    i = -1

    for seq1 in guide[: -1]:
        i += 1
        seq2 = guide[i + 1]
        name = (seq1
            + '_'
            + seq2)
#cofold is a ViennaRNA package function that calculates
#the complex' MFE and structure
        (ss, mfe) = (RNA.cofold(seqs[seq1]
            + '&'
            + seqs[seq2])

        seq_DG[name] = mfe
        seq_ss[name] = (ss[:len(seqs[seq1])]
            + '&'
            + ss[(len(seqs[seq1])):-1])

    P1 = bolfunc(guide[0], 'sensor', seq_DG)
    P2 = bolfunc('transducer', 'clamp', seq_DG)
    P3 = probfunc('sensor', 'transducer', seq_DG, seqs)
    P4 = probfunc('clamp', 'T7p', seq_DG, seqs)
    T = toeholdscore('sensor_transducer', seq_ss)

    score = P1*P2*P3*P4*(6-T)/6  #The Objective Function
    dats = [P1,P2,P3,P4,T,score]
    return dats
```

Box 2*: Python functions for Score Function calculus*

The optimization consists in the calculation of the Objective Function (***Eq. (6)***) prior to any mutation and after a random base substitution mutation on a random component of the circuit (different from the miRNA and T7p). If the mutation favors the Objective Function, the mutation is kept, while if it doesn't, the mutation is rejected. The code implementation is the following:

```
#Define nucleotides
NUCS = ['A','T','G','C']
#Define mutation function
def mutf(seqs):
    seqs_aftermutation = {}

    #Creates a new dictionary with sequences
    for element in seqs:
        seqs_aftermutation[element] = seqs[element]

```

This box continues on the next page

```
    #Chooses a random base from a random sequence
    target_name = random.sample(guide[1:4], 1)[0]
    target_seq = list(seqs[target_name])
    position = random.randint(0, (len(target_seq) - 1))
    base = random.sample(NUCS, 1)[0]

    while base == target_seq[position]:
        base = random.sample(NUCS, 1)[0]

    #Writes the mutated sequence
    target_seq[position] = base
    target_seq = ''.join(target_seq)
    seqs_aftermutation[target_name] = target_seq

    return seqs_aftermutation

def main():
    global k, timesuffix, seqs_preit, seqs_posit
    global Score_preit, Score_posit, Dats_preit, Dats_posit

    #Moment in time:
    timesuffix = '_'.join(
        str(datetime.datetime.now()
        ).split())

    (seqs_preit['sensor'],
        seqs_preit['transducer'],
        seqs_preit['clamp'],
        seqs_preit['fuel']) = genseq(seqs_preit[GUIDE[0]], seqs_preit['T7p'])

    Dats_preit = scorefunc(seqs_preit)
    Score_preit = Dats_preit[-1]

#100000 cycles of mutations and selection following the global score
    k = 0
    for n in range(int(1e5)):
        k += 1
        seqs_posit = mutf(seqs_preit)
        Dats_posit = scorefunc(seqs_posit)
        Score_posit = Dats_posit[-1]

        if Score_posit >= Score_preit:
            Dats_preit = Dats_posit
            Score_preit = Score_posit
            seqs_preit = seqs_posit

    OUTFILE = open(('Output_'
        + guide[0]
        + timesuffix
        + '.txt'),
        'w')
    OUTFILE.write('This is the output of your job done on '
        + timesuffix
        + '\n')
```

This box continues on the next page

```
    for el in guide:
        OUTFILE.write('>'
            + el
            + '\n'
            + seqs_preit[el]
            + '\n')
    OUTFILE.write('\nP1 = '+ str(Dats_preit[0]) + '\n')
    OUTFILE.write('P2 = ' + str(Dats_preit[1]) + '\n')
    OUTFILE.write('P3 = ' + str(Dats_preit[2]) + '\n')
    OUTFILE.write('P4 = ' + str(Dats_preit[3]) + '\n')
    OUTFILE.write('Toehold = ' + str(Dats_preit[4]) + '\n')
    OUTFILE.write('Score = ' + str(Score_preit) + '\n')
    OUTFILE.close()

    return None
```

***Box 3**: Python functions for sequence mutation and score selection*

3.5 Metropolis Algorithm implementation

The risk of rejecting all mutations that do not favor the Objective Function is that a possible absolute maximum value could be missed due to a valley of unfavorable values that may be surrounding this maximum in the space of probabilities. To allow a “local scanning” in the space of probabilities, this algorithm is executed whenever a mutation is rejected.

To do so, a “Metropolis factor” is calculated the following way:

$$M = e^{-\beta_M^0 \delta^t (F_{Score} - F_{Score}^*)} \qquad \text{Eq.}\,(7)$$

Where:

β_M^0: initial factor that defines a probability of 0.01 of accepting an unfavorable mutation $\approx$ 1100

δ: a factor representing a decrease of the probability of accepting an unfavorable mutation = 1.00007

t: iteration number

F_{Score}: Objective function value prior to iteration

F^*_{Score}: Objective function value after iteration

Next, the Metropolis factor (***Eq. (7)***) is compared against a random generated number ranging from 0 to 1. If the Metropolis factor is higher than this value, the unfavorable mutation is accepted. If not, it is rejected.

The idea is that the more detrimental the mutation is, the lower the Metropolis factor (***Eq. (7)***), and therefore the probability of it being below the random generated number is higher.

The metropolis algorithm is implemented in the main code as a function, which is executed under an `else` statement just after the `if Score_posit >= Score_preit` statement shown in ***Box 3***.

The Metropolis function is illustrated in ***Box 4***:

```
#Metropolis parameters
Bm0 = 1100
D = 1.00007  #delta

def Metropolis():
    global Dats_preit, Score_preit, seqs_preit
    Bmk = Bm0*(D**k)
    M = NUM_e**(
        - Bmk*(
            Score_preit
            - Score_posit))

    if random.random() < M:
        Dats_preit = Dats_posit
        Score_preit = Score_posit
        seqs_preit = seqs_posit
    return None
```

Box 4*: Python Metropolis function*

3.6 Simple Circuit Kinetic Model Design

The system to be modeled can be easily described with the following reactions:

$$\begin{cases} m \ + \ s{:}t \xrightarrow{k_S} m{:}s \ + t \\ t \ + \ cl{:}TS \xrightarrow{k_E} t{:}cl \ + TS \end{cases} \qquad \text{Eq. (8)}$$

Where:

m : free miRNA concentration (μM)

s:t : sensor-transducer complex concentration (μM)

k_S : transducer liberation kinetic constant (μM s^{-1})

m:s : miRNA-sensor complex concentration (μM)

t : free transducer concentration (μM)

cl:TS : clamp-T7p complex concentration (μM)

k_E : T7p liberation kinetic constant (μM s^{-1})

t:cl : transducer-clamp complex concentration (μM)

TS : free T7p concentration (μM)

The kinetic constants of the reactions illustrated in ***Eq. (8)*** are unknown as the main method of determining strand displacement reaction kinetic constants is by means of experimental measures. Nevertheless, Zhang & Winfree (2009), in an attempt to model the kinetic constants of these reactions, presented a simple flowchart which by taking into account toehold length (n) and reaction mechanism (toehold mediated strand displacement or toehold exchange) indicates an approximation of the kinetic constants for each individual reaction. For both reactions presented previously, the mechanism considered is toehold mediated strand displacement (having a value of m = 0 regarding

Zhang & Winfree's flowchart) and both toeholds employed have a length of 6 nucleotides (n = 6) (***Table 2***). This data concludes that the values of k_s and k_E is $0.5\ \mu M^{-1}s^{-1}$.

As this system consists in two reactions, where the second one is limited by the species produced on the first one, in order to describe the rate of T7p liberation, the rate of transducer liberation has to be taken into account as well. For that purpose, the following differential equations were inferred from the reactions:

$$\begin{cases} \dfrac{dt}{d\tau} = k_s \cdot m \cdot s{:}t \\ \dfrac{dTS}{d\tau} = k_E \cdot t \cdot cl{:}TS \end{cases} \qquad \text{Eq. (9)}$$

Where:

$\frac{dt}{d\tau}$: rate of transducer liberation (μM s^{-1})

$\frac{dTS}{d\tau}$: rate of T7p liberation (μM s^{-1})

τ : time (s)

Furthermore, a mass balance of species has to be taken into account:

$$\begin{cases} m_{Total} = m + m{:}s \\ t_{Total} = t + s{:}t \\ TS_{Total} = cl{:}TS + TS \end{cases} \qquad \text{Eq. (10)}$$

Where:

i_{Total} : the total amount of species "i", either free or not (μM)

The inclusion of ***Eq. (10)*** into ***Eq. (9)*** yields:

$$\begin{cases} \dfrac{dt}{d\tau} = k_s \cdot (m_{Total} - m{:}s) \cdot (t_{Total} - t) \\ \dfrac{dTS}{d\tau} = k_E \cdot t \cdot (TS_{Total} - TS) \end{cases} \qquad \text{Eq. (11)}$$

In addition, the next considerations can be done: as species *m:s* and species *t* are generated in the same reaction, at the same rate and amount, they can be considered equal; prior to further tweaking, this first model will consider that all total amounts of species are equal (which means that there is the same concentration of each circuit component and miRNA, being this value 1 μM). This yields the following expression:

$$\begin{cases} \dfrac{dt}{d\tau} = k_s \cdot (c - t)^2 \\ \dfrac{dTS}{d\tau} = k_E \cdot t \cdot (c - TS) \end{cases} \qquad \text{Eq. (12)}$$

Where:

c : the total concentration of each species (1 μM)

At this point, ***Eq. (12)*** can easily undergo analytical integration:

$$\int_0^{t(\tau)} \frac{dt}{(c-t)^2} = \int_0^{\tau} k_s d\tau\,;$$

$$t = \frac{k_s \cdot c^2 \cdot \tau}{1 + k_s \cdot c \cdot \tau} \qquad \text{Eq. (13)}$$

Going back to ***Eq. (11)***, the value for t can be substituted, yielding:

$$\frac{dTS}{d\tau} = k_E \cdot \left(\frac{k_s \cdot c^2 \cdot \tau}{1 + k_s \cdot c \cdot \tau}\right) \cdot (TS_{Total} - TS) \qquad \text{Eq. (14)}$$

Eq. (14) can be subjected to analytical integration:

$$\int_0^{TS(\tau)} \frac{dTS}{(TS_{Total} - TS)} = k_E \int_0^{\tau} \left(\frac{k_s \cdot c^2 \cdot \tau}{1 + k_s \cdot c \cdot \tau}\right) d\tau\,;$$

$$TS = c \cdot \left(1 - \frac{(1 + k_s \cdot c \cdot \tau)^{k_E/k_s}}{e^{k_E \cdot c \cdot \tau}}\right) \qquad \text{Eq. (15)}$$

Taking into account that in this particular case $k_s = k_E$, ***Eq. (15)*** can be further simplified:

$$TS = c \cdot \left(1 - \frac{1 + k_s \cdot c \cdot \tau}{e^{k_E \cdot c \cdot \tau}}\right) \qquad \text{Eq. (16)}$$

It is important to note that ***Eq. (16)*** overestimates the catalytic capacity of the circuit as it assumes ideal conditions, kinetic constants and concentrations.

In addition, it should be noted again that for the sake of simplicity, the kinetic model assumed that all species (including the input miRNA) are at the same concentration, which is the ideal situation. But as this case is very rare, since miRNA concentrations in biological samples are very small, a redesign of the circuit is necessary to ensure amplification of a signal originating from a tiny amount of input, which in turn adds complexity to its corresponding kinetic model.

3.7 Signal Amplification Circuit Components

Similar to the simple circuit in ***Figure 4***, the initial state consists in an equilibrium in which a single stranded molecule named fuel co-exists along the stable and pre-formed complexes sensor-transducer and clamp-T7p.

The addition of miRNA triggers the circuit in exactly the same manner as the simple circuit (***Figure 4***), but with the exception that fuel will displace miRNA from the miRNA-sensor complexes, forming fuel-sensor complexes.

The purpose of this additional reaction is to liberate miRNA that might further react with sensor-transducer complexes, generating therefore a cyclic signal amplification, as illustrated in ***Figure 6***.

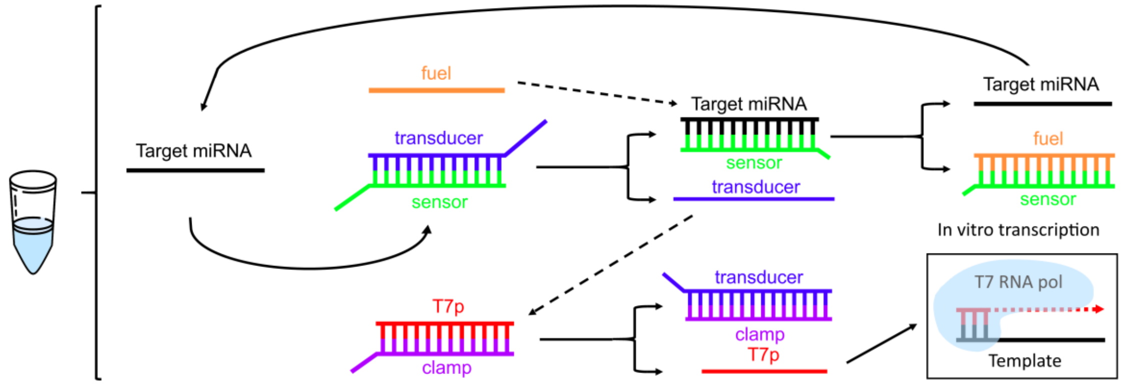


Figure 6: *Signal amplification circuit components and interactions*

3.8 Signal Amplification Circuit Initial Sequence Design

The approach is inherited from the simple circuit design, but with a particularity: the master sequence includes now a spacer of 5 nucleotides between the joining point of the target miRNA sequence and T7p. The majority of the components become elongated due to these 5 additional nucleotides, except clamp, whose elongation is avoided on purpose to avoid its toehold elongation.

These 5 additional nucleotides will be part of the toehold that will promote fuel-sensor formation and miRNA displacement and are randomly generated each time the algorithm is executed. An example is shown below:

Table 3: *Signal amplification circuit components initial sequence generation example*

```
Master:           TGGAGTGTGACAATGGTGTTTGNNNNNGCGCTAATACGACTCACTATAGG
miRNA (5’-3’):    TGGAGTGTGACAATGGTGTTTG
sensor (3’-5’):   ACCTCACACTGTTACCACAAACNNNNNCGC
transducer (5’-3’):     GTGACAATGGTGTTTGNNNNNGCGCTAATACGACTCACTATAGG
clamp (3’-5’):                         CNNNNNCGCGATTATGCTGAGTGATATCC
T7p (5’-3’):                                 GCGCTAATACGACTCACTATAGG
fuel (5’-3’):           GTGACAATGGTGTTTGNNNNNGCG
```

The implementation in code is shown in ***Box 5***:

```
#Random sequence builder
def randseq(length):
    out = ''
    for n in range(length):
        out += random.sample(NUCS, 1)[0]
    return out

#Define circuit core sequences generator
def genseq(miRNA, prom):
    n = len(miRNA)
    rootseq = (miRNA.upper()
        + randseq(5)
        + prom)
    sensor = revcomp(
        rootseq[: n+8])
    transducer = rootseq[6:]
```

This box continues on the next page

```
    clamp = revcomp(
        rootseq[n - 1 :])
    fuel = rootseq[6: n + 8]
    return (sensor,
        transducer,
        clamp,
        fuel)
```

Box 5: *Python functions for signal amplification circuit initial sequence generation*

3.9 Adaptation of the Objective Function

The addition of a new species, and therefore a new reaction, to the simple circuit (***Figure 4***) forces a modification of the objective function (***Eq. (6)***) employed for the circuit's scoring. It is necessary the addition of a term that takes into consideration the probability of fuel-sensor complex formation. There is, however, a risk in favoring the formation of fuel-sensor complex as it may cause an erroneous behavior of the circuit since fuel may act as input, which is undesired. Nevertheless, this event doesn't have the tendency to occur as, although fuel-sensor may have a lower MFE than sensor-transducer, the complex sensor-transducer is pre-formed and lacks the toehold that initiates the formation of fuel-sensor (which was taken into account during the sequence design).

Therefore, fuel will only interact with miRNA-sensor complex and form sensor-transducer because it has the toehold that allows its interaction with miRNA-sensor and the complex fuel-sensor has a lower MFE than miRNA-sensor, causing this reaction to occur spontaneously.

In a similar fashion as presented for the other components of the circuit, the probability of fuel-sensor complex formation is the following:

$$P_5 = \frac{e^{-\beta\Delta G_{fuel-sensor}}}{e^{-\beta\Delta G_{fuel-sensor}} + e^{-\beta\Delta G_{miRNA-sensor}}} \qquad \text{Eq. (17)}$$

The modified Objective Function that considers the ***Eq. (17)*** is illustrated in ***Eq. (18)***:

$$F_{score} = P_1 P_2 P_3 P_4 P_5 \left(\frac{6-T}{T}\right) \qquad \text{Eq. (18)}$$

Eq. 18 will be employed during the optimization, just as previously mentioned in the simple circuit. The corresponding code is a simple tweak from the code presented in ***Box 6***, as it can be seen below:

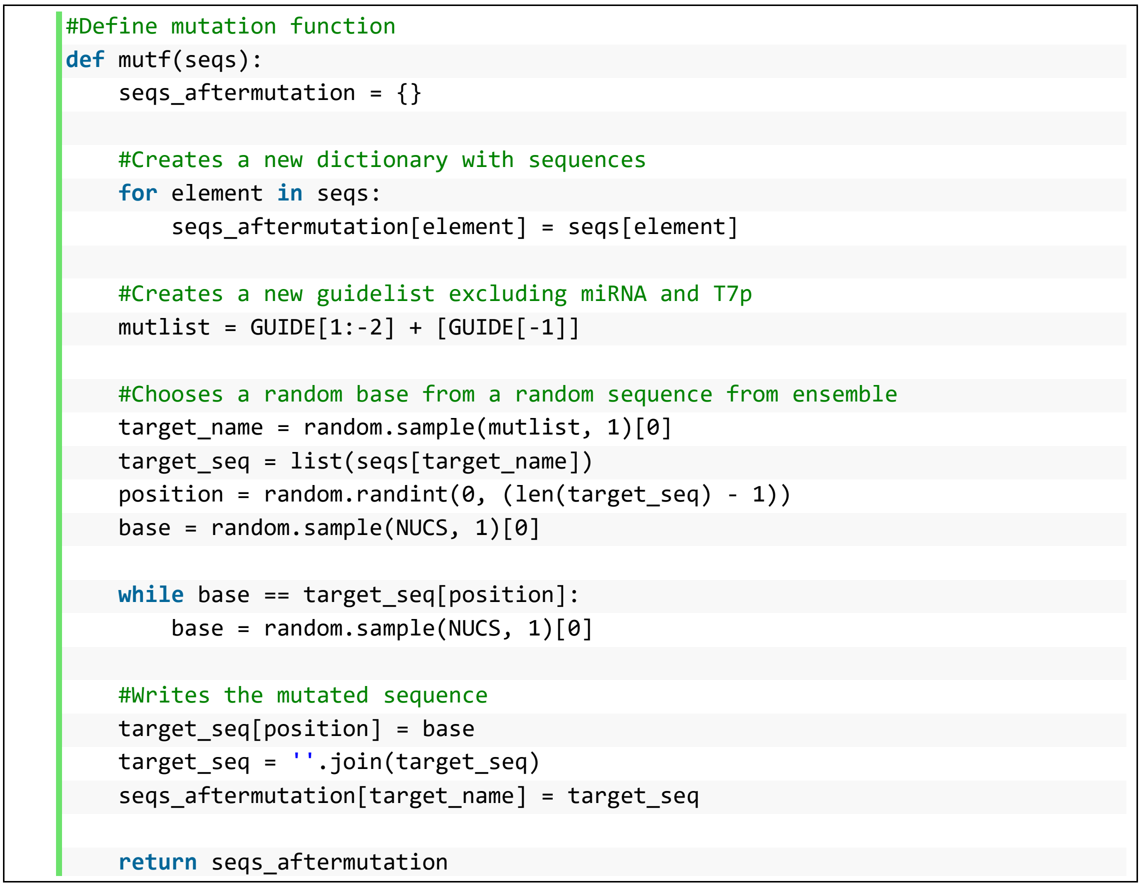
```
#Define circuit sequence names
GUIDE = ['miRNA', 'sensor', 'transducer', 'clamp', 'T7p', 'fuel']
#Define Boltzmann function
def bolfunc(seq1, seq2, seq_DG):
    Pairkey = (seq1
        + '_'
        + seq2)

    Numerator = NUM_e**(- BETA*seq_DG[Pairkey])
    Denominator = Numerator

    if seq1 == GUIDE[0]:
        SecondKey = 'sensor_transducer'
```

This box continues on the next page

```
    elif seq1 == 'transducer':
        SecondKey = 'clamp_T7p'

    elif seq1 == 'fuel':
        SecondKey = GUIDE[0] + '_sensor'

    Denominator += NUM_e**(- BETA*seq_DG[SecondKey])
    func = Numerator/Denominator

    return func

#Define Packing and Scoring function.
def scorefunc(seqs):
    seq_DG = {}
    seq_ss = {}
    i = -1
    #Saves in a dictionary the MFE and structure of circuit pairs
    for seq1 in GUIDE[:-2]:
        i += 1
        seq2 = GUIDE[i + 1]
        name = (seq1
            + '_'
            + seq2)
        (ss, mfe) = RNA.cofold(
            (seqs[seq1]
            + '&'
            + seqs[seq2]))
        seq_DG[name] = mfe
        seq_ss[name] = (ss[: len(seqs[seq1])]
            + '&'
            + ss[(len(seqs[seq1])) :-1])

    (ss, mfe) = RNA.cofold(
        (seqs['fuel']
        + '&'
        + seqs['sensor']))
    seq_DG['fuel_sensor'] = mfe
    seq_ss['fuel_sensor'] = (ss[: len(seqs['fuel'])]
        + '&'
        + ss[(len(seqs['fuel'])) :-1])
    #Calculates pair probabilities and Score

    P1 = bolfunc(GUIDE[0], 'sensor', seq_DG)
    P2 = bolfunc('transducer', 'clamp', seq_DG)
    P3 = probfunc('sensor', 'transducer', seq_DG, seqs)
    P4 = probfunc('clamp', 'T7p', seq_DG, seqs)
    P5 = bolfunc('fuel', 'sensor', seq_DG)
    T = toeholdscore('sensor_transducer', seq_ss)

    score = P1*P2*P3*P4*P5*(6-T)/6
    dats = [P1,P2,P3,P4,P5,T,score]
    return dats
```

***Box 6**: Modified Python functions for Score Function calculus*

The addition of a new species that can be subjected to mutation forces a modification in the mutation function code presented in ***Box 7***:



```
#Define mutation function
def mutf(seqs):
    seqs_aftermutation = {}

    #Creates a new dictionary with sequences
    for element in seqs:
        seqs_aftermutation[element] = seqs[element]

    #Creates a new guidelist excluding miRNA and T7p
    mutlist = GUIDE[1:-2] + [GUIDE[-1]]

    #Chooses a random base from a random sequence from ensemble
    target_name = random.sample(mutlist, 1)[0]
    target_seq = list(seqs[target_name])
    position = random.randint(0, (len(target_seq) - 1))
    base = random.sample(NUCS, 1)[0]

    while base == target_seq[position]:
        base = random.sample(NUCS, 1)[0]

    #Writes the mutated sequence
    target_seq[position] = base
    target_seq = ''.join(target_seq)
    seqs_aftermutation[target_name] = target_seq

    return seqs_aftermutation
```

***Box 7**: Modified Python mutation function*

3.10 Signal Amplification Circuit Kinetic Model Design

In this case, the system to be modeled has a higher complexity as it takes into account an additional reaction. Furthermore, approximations regarding total concentrations of the components cannot be performed as this circuit's purpose is to amplify a very low miRNA input signal, thus it is interesting to elaborate a model that works with varying total miRNA concentrations:

$$\begin{cases} m \ + \ s{:}t \overset{k_S}{\rightarrow} m{:}s \ + t \\ t \ + \ cl{:}TS \overset{k_E}{\rightarrow} t{:}cl \ + TS \\ f \ + \ m{:}s \overset{k_F}{\rightarrow} f{:}s \ + m \end{cases} \qquad \text{Eq. (19)}$$

Where:

f : free fuel concentration (μM)

f:s : fuel-sensor complex concentration (μM)

k_F : miRNA liberation kinetic constant (μM s^{-1})

The kinetic constants of the reactions inherited from the simple model remain the same as estimated previously by means of the flowchart provided by Zhang & Winfree (2009). To estimate the value of k_F, the same approach was employed. In this case, the toehold length is of 8 nucleotides (n = 8) (***Table 3***) while the mechanism remains the same (m = 0), yielding a value of approximately $3\ \mu M^{-1}s^{-1}$. In order to describe the evolution of all the system's components with time, the following expressions were inferred:

$$\begin{cases} \dfrac{dm}{d\tau} = -k_s \cdot m \cdot s{:}t + k_F \cdot f \cdot m{:}s \\ \dfrac{ds{:}t}{d\tau} = -k_s \cdot m \cdot s{:}t \\ \dfrac{dt}{d\tau} = k_s \cdot m \cdot s{:}t - k_E \cdot t \cdot cl{:}TS \\ \dfrac{dm{:}s}{d\tau} = k_s \cdot m \cdot s{:}t - k_F \cdot f \cdot m{:}s \\ \dfrac{dcl{:}TS}{d\tau} = -k_E \cdot t \cdot cl{:}TS \\ \dfrac{dTS}{d\tau} = k_E \cdot t \cdot cl{:}TS \\ \dfrac{dt{:}cl}{d\tau} = k_E \cdot t \cdot cl{:}TS \\ \dfrac{df}{d\tau} = -k_F \cdot f \cdot m{:}s \\ \dfrac{df}{d\tau} = k_F \cdot f \cdot m{:}s \end{cases} \qquad \text{Eq. (20)}$$

For a better understanding of the model, ***Eq. 20*** can be simplified, yielding:

$$\begin{cases} \dfrac{dm}{d\tau} = \dfrac{ds{:}t}{d\tau} - \dfrac{df}{d\tau} \\ \dfrac{ds{:}t}{d\tau} = -k_s \cdot m \cdot s{:}t \\ \dfrac{dt}{d\tau} = -\left(\dfrac{ds{:}t}{d\tau} + \dfrac{dTS}{d\tau}\right) \\ \dfrac{dm{:}s}{d\tau} = -\dfrac{dm}{d\tau} \\ \dfrac{dcl{:}TS}{d\tau} = -\dfrac{dTS}{d\tau} \\ \dfrac{dTS}{d\tau} = k_E \cdot t \cdot cl{:}TS \\ \dfrac{dt{:}cl}{d\tau} = \dfrac{dTS}{d\tau} \\ \dfrac{df}{d\tau} = -k_F \cdot f \cdot m{:}s \\ \dfrac{df{:}s}{d\tau} = -\dfrac{df}{d\tau} \end{cases} \qquad \text{Eq. (21)}$$

In order to ease the modelling procedure, the previous system of differential equations was integrated numerically by means of the tool "odeint" provided by SciPy python library (SCIPY, 2019). In addition, parameters such as miRNA and fuel total concentration were modified with the aim of characterizing the system's behavior and finding the most suitable fuel concentration for the circuit to operate efficiently, plus discovering the miRNA concentration threshold for which the circuit would act as a viable alternative for miRNA detection.

3.11 Leakage Prevention Strategy

As mentioned previously in this work, one of the main disadvantages of synthetic DNA circuits is a spontaneous activation of the circuit in absence of input signal. This phenomenon is known as leakage and it is caused by spontaneous fluctuations in hybridization between strands due to temperature. Recently, two strategies that attempt to cope with signal leak have become popular among synthetic biologists. The first strategy, proposed by Wang *et al.* (2018) is to incorporate in the circuit's design a series of components that, similarly to the strategies employed in electrical engineering, act as redundant blocks which in order to leak signal require a sequence of energetically unfavorable events to happen, thus reducing leak occurrence. The second strategy, proposed by Song *et al.* (2018), consists in the elaboration of a parallel circuit, with similar characteristics as the main circuit, that works "in the shadow" of the main circuit, which would leak signal at a similar rate than the main circuit. Both leaks are sequestered by an AND gate, therefore the presence of signal in the absence of the shadow circuit's leak won't get silenced. Although the shadow circuit's leak would be constantly causing signal loss (at a rate proportional to the shadow circuit's leak), signal produced by presence of input should occur at such a higher rate that the effect of the shadow circuit would be negligible.

The latter strategy is considered most suitable for its application on this work's circuit, as it does not require a complete re-design of the circuit's components (***Figure 6***). It was considered that the main source of signal leak in the circuit was a spontaneous dissociation of sensor-transducer, generating a free transducer that would displace T7p from clamp-T7p complex, generating signal. An example of a proposed leaked signal silencing is showed below:

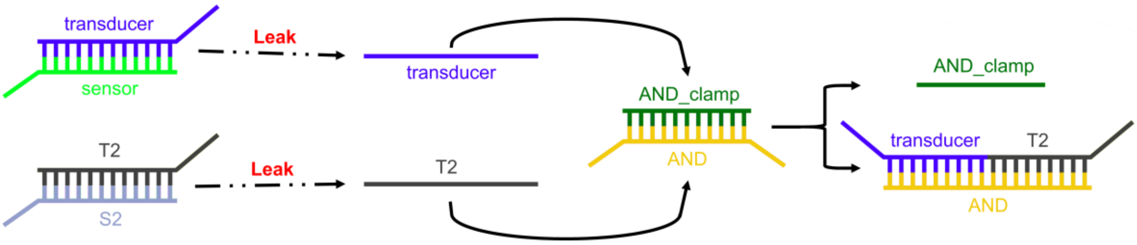


Figure 7*: Leaked signal silencing by shadow circuit*

As seen in ***Figure 7***, the shadow circuit would consist on species S2, T2, AND & AND_clamp. The initial sequence design for the shadow circuit depends on the sequence of the transducer, as it is essential for the design of AND & AND_clamp species. S2 and T2 sequences are obtained from the sequences employed by Song *et al.* (2018) in their own work and adapted for each circuit as complexes S2-T2 and sensor-transducer should leak signal in a similar manner.

To accomplish that purpose, it is quite a good approximation to assume that their MFEs should be equal (if not, similar) as signal leak depends on spontaneous strand dissociation due to energy fluxes, which may cause the strands to overcome the energy barrier that impedes them to break the complex.

Therefore, S2 and T2 are subjected to a round of guided evolution prior to the shadow circuit's sequence design in which their MFE is compared with the corresponding sensor-transducer MFE. If said MFE is lower than the corresponding sensor-transducer MFE, a mutation substituting a random C-G (or G-C) pair for a A-T (or T-A) pair is performed in the S2-T2 interaction site, which reduces the complex’ MFE. If the MFE would be lower, the contrary action is performed.

This simple approach is effective as the complex’ MFE is solely dependent on salt concentration in the media and base composition. In the case where sensor-transducer complex’ MFE was lower than -31kcal/mol, the S2-T2 binding sites would be enlarged systematically (adding bases in a random manner), to enlarge the number of base pairs that contribute towards the complex’ MFE. Once S2-T2 and sensor-transducer MFEs are equal, it is safe to proceed towards the circuit’s sequence generation.

Similar as performed for the main circuit, the generation AND & AND_clamp species is done by means of a master sequence, which in turn is elaborated through the concatenation of the last 20 nucleotides of T2 sequence and the first 19 nucleotides of the transducer sequence. An example of sequence design is shown in ***Table 4***:

```
Master:                 CATCTCAAACACTCTATTCAGTGACAATGGTGTTTGGCG
Transducer(5’-3’):                          GTGACAATGGTGTTTGGCGCTAAT…
AND(3'-5'):             GTAGAGTTTGTGAGATAAGTCACTGTTACCACAAACCGC
AND_clamp(5’-3’):              AACACTCTATTCAGTGACAATGGTGT
T2(5’-3’): CACTCATCCTTTACATCTCAAACACTCTATTCA
```

Table 4: *Shadow circuit sequence design example*

The sequence design is implemented in code as shown in ***Box 8***:

```
#Define shadow circuit constant components
shdw = {'S2': 'TGAGATGTAAAGGATGAGTGAGATG',
    'T2': 'CACTCATCCTTTACATCTCAAACACTCTATTCA'}

#Define shadow circuit generation function
def shadowcirc(transducer):
    outdict = {}

    for el in shdw:
        outdict[el] = shdw[el]

    MFE = RNA.cofold(
        seqs_preit['sensor']
        + '&'
        + seqs_preit['transducer'])[1]

    mfe = RNA.cofold(
        outdict['S2']
        + '&'
        + outdict['T2'])[1]

    b_area = outdict['S2'][:-5]

    if MFE < -31:
        times = int((MFE + 31)/3) + 4

        for n in range(times):
            b_area += random.sample(NUCS, 1)[0]

    while abs(MFE - mfe) > 0:

        target_index = random.randint(0, (len(b_area) - 1))
```

This box continues on the next page

```
        b_area = list(b_area)
        base = random.sample(NUCS, 1)[0]

        while base == b_area[target_index]:
            base = random.sample(NUCS, 1)[0]

        b_area[target_index] = base
        b_area = ''.join(b_area)

        outdict['S2'] = (b_area
            + outdict['S2'][-5:])

        outdict['T2'] = (revcomp(b_area)
            + outdict['S2'][-13:])

        mfe = RNA.cofold(
            outdict['S2']
            + '&'
            + outdict['T2'])[1]

        for el in NUCS:
            if (4*el) in b_area:
                mfe = 1e3

    master = (outdict['T2'][-20:]
        + transducer[:19])

    AND_clamp = master[7:-6]
    AND = revcomp(master)

    outdict['AND_clamp'] = AND_clamp
    outdict['AND'] = AND

    keyss = []
    for el in outdict.keys():
        keyss += [el]
    keyss.sort()

    return outdict, keyss
```

Box 8: *Python functions for shadow circuit sequence design*

4 Results and Discussion

4.1 Score Function convergence

As mentioned in Materials and methods, the mutations and selection to which the circuit is subjected have the objective of maximizing the Score by, in turn, maximizing each of the terms that compose the Score. The term $P4$, however, does never reach a value close to 1, mainly because of 2 reasons: the complementarity between clamp and T7p is at its maximum from the beginning, thus every mutation that would affect clamp is from the beginning detrimental and the main function of the term $P4$ in the Score Function is to avoid the algorithm from increasing the $P2$ term by lowering the Boltzmann function of clamp-T7p, which increases the Score Function. Although the term $P3$ behaves in a similar manner as the term $P4$ (as it has a similar function), it reaches a maximum value of 1 because the Boltzmann function for sensor-transducer is easily higher than the artificial Boltzmann function which it compares to, allowing in this case the occurrence of mutations in sensor and transducer as long as their complex' Boltzmann function overcomes the artificial threshold.

In order to obtain a Score from which the quality of the circuit can be interpreted, and considering that in every experimental run of the algorithm the value of $P4$ does not change from iteration 0 to iteration 100000, a standardized score can be calculated by dividing the value of the score by the value of $P4$ and multiplying by 100. Note that this standardized score is not employed during the selection step as it does not conserve the contribution of $P4$ towards the score.

To observe how the score value approaches a maximum with the given Metropolis parameter $\beta_M^0 \approx 1100$ during the runtime of the algorithm, ***Figure 8*** was elaborated using the standardized score.

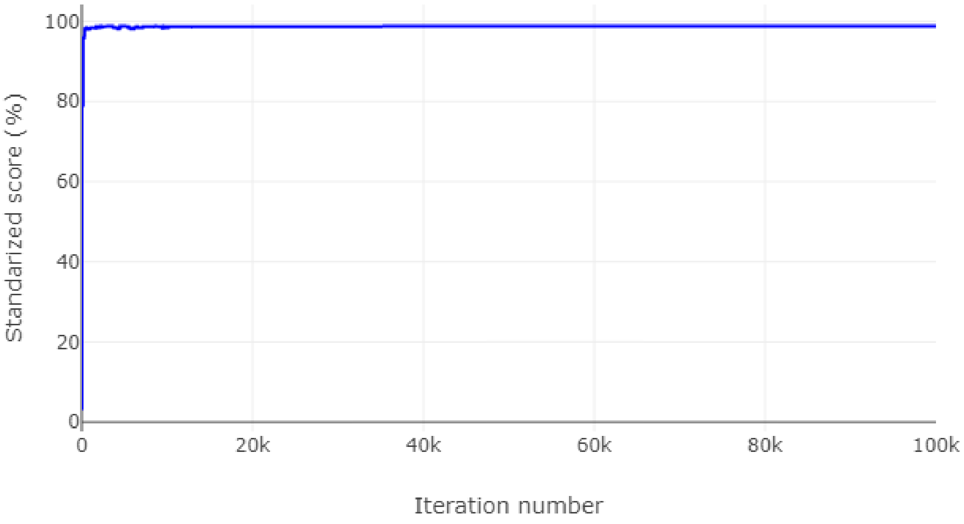


Figure 8*: Score convergence during algorithm run*

As seen above, the algorithm fluctuates until reaching a maximum value approaching 100, being that value in this case 98.8364. To observe in detail how the score progresses, a zoom in is made, resulting in ***Figure 9***:

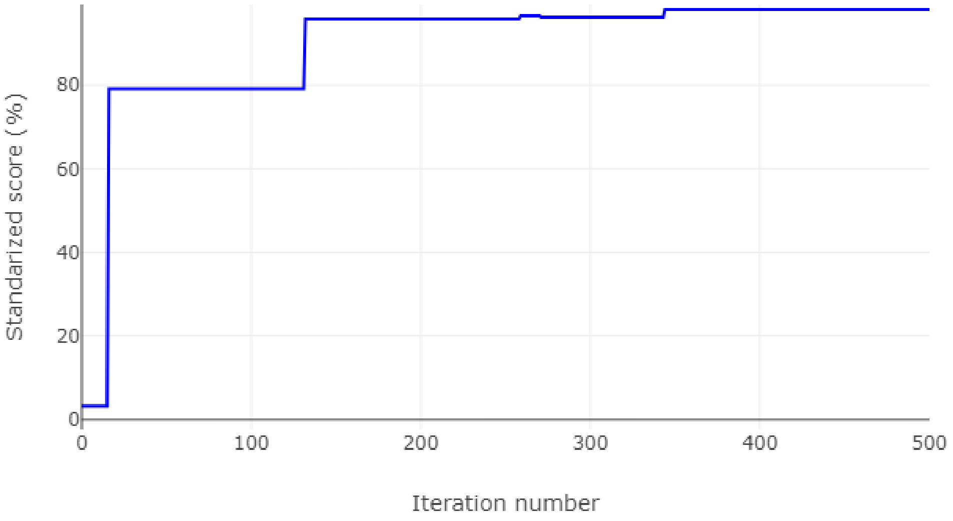


Figure 9*: Score convergence during the first 500 iterations*

The initial score is 3.2060, which increases steeply during the first iterations due to single mutations. This is feasible as there are base pairs that have a higher impact on the MFE of complexes as they may heavily affect the structure through forces of repulsion or attraction. Additionally, it can be observed that from time to time, the score gets reduced, which is result of accepting an unfavorable mutation due to the Metropolis function. Nevertheless, several iterations later, a single mutation achieves to increase the score, fact that could have not happened without the previous unfavorable mutation. Therefore, it is safe to say that the randomicity provided by the Metropolis function during early iterations indeed enables the Score to explore a wider space of probabilities, avoiding getting stuck at local maximums, while restricting at higher iterations the loss of the maximum encountered.

4.2 Metropolis effect on Score

Although there is evidence that the Metropolis function and its proposed parameters contribute in the randomization of the selection without being detrimental, it is not directly known how the modification of its parameters would affect the algorithm.

The effect of δ (***Eq. (7)***) is straightforward to foresee, as it represents a reduction in the probability of accepting a detrimental mutation. If this term would be equal to 1, the Metropolis Function would act as a constant threshold, therefore the probability of accepting a detrimental mutation would be only determined by the random number generation and the value of β_M^0. If the term would be lower than 1, it would increase the probability of accepting a detrimental mutation, causing the Score not to converge towards a maximum. If the term would be much higher than 1, randomicity would not be evenly distributed along the iteration numbers, meaning that there would only be randomicity during the first 5 iterations while being absent during the remaining 9995 iterations (for example). As the effect of δ is so sensitive to small changes, it is better not to rely on it to control the effect of the Metropolis function.

On the other hand, the effect of β_M^0 (***Eq. (7)***) is less clear. It defines the initial probability of accepting a detrimental mutation, which is reduced with each iteration by means of the δ constant. However, it is unknown how the algorithm behaves under different values of β_M^0. With the purpose of analyzing this effect, the Score convergence was studied in a similar manner as done previously in this work, but with different values for β_M^0 while maintaining constant the input miRNA, which is the same as the one employed in the sequence design step (***Table 2*** & ***Table 3***). The resulting figure is shown below:

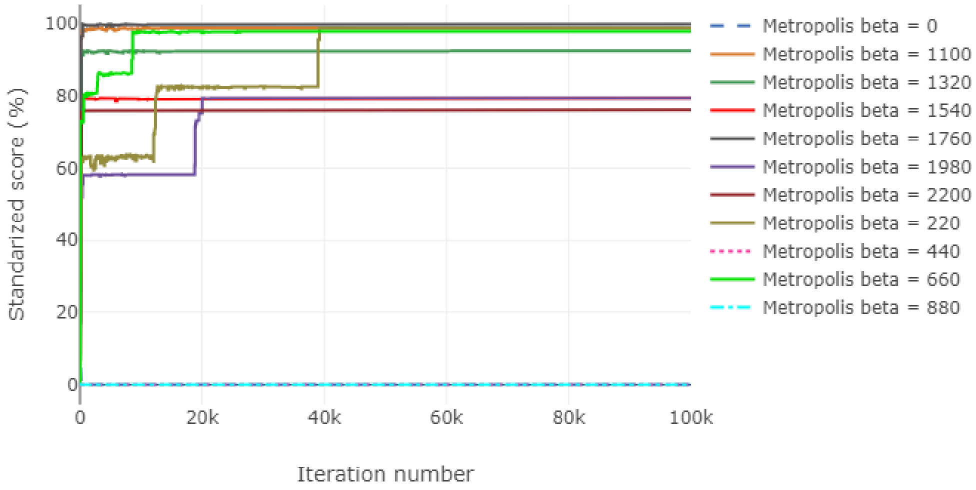


Figure 10*: Metropolis beta effect on Score convergence*

It can be observed that the β_M^0 values below the proposed constant $\beta_M^0 = 1100$ have the general effect of avoiding Score convergence towards a value approaching 100 but sinking the score towards 0. There is, however, the exception of $\beta_M^0 = 220$ and $\beta_M^0 = 660$, as these two β_M^0 values allowed the Score to converge close to 100. The reason behind these behaviors is that a lower β_M^0 value increases the probability of accepting a detrimental mutation, which may allow that various detrimental mutations in a row accumulate, sinking the score significantly. In addition, because of the Metropolis function getting more and more stringent as iterations increase, the algorithm will not be able to recover from this low score, therefore getting stuck at a minimum, which is usually 0. The excellent performance of $\beta_M^0 = 220$ and $\beta_M^0 = 660$ can be explained by means of two phenomena: a good initial Score and single mutations whose effect on the complex' MFE is vastly favorable. The initial Score of the circuit depends exclusively on the circuit sequence design step, in which there are 5 nucleotides that are generated randomly each time the algorithm is executed (***Table 3***), so by having a better initial Score, the stability of the complexes is higher, which means that they may suffer a couple detrimental mutations in a row without dramatically sinking the score. On the other hand, the effect of single mutations with great effect on MFE depends exclusively on luck, as mutations are completely random, and chances are higher that mutations are detrimental rather than favorable, that is why there is a need for a β_M^0 that limits the amount of detrimental mutations accepted by the algorithm.

On the other hand, β_M^0 values above the proposed constant $\beta_M^0 = 1100$ have the effect of trapping the algorithm on local maximums if the right mutations do not occur, as there is a reduced randomicity, which means a reduced ability to explore the space of probabilities. This effect can be clearly observed with β_M^0 values 1320, 1540, 1980 and 2200, whose final scores are stuck at 92.62, 79.39, 79.40 and 76.14 respectively. This effect is not observed with $\beta_M^0 = 1760$, which has a final score of 99.88. This exception is probably due to encountering the appropriate mutations that directly sent the score towards the maximum, an event which has very low probabilities to occur if the "local scanning" of the space of probabilities is reduced with such value of β_M^0. What was expected from $\beta_M^0 = 1760$ is to get stuck at a similar score than $\beta_M^0 = 1320$ and $\beta_M^0 = 1540$.

It is very important to remark that the exceptions encountered during this analysis are not exclusive to their associated β_M^0 values, which means that in a repetition of the experiment, there might be other values β_M^0 which suffer these exceptions as the explanations provided can be applied to every β_M^0 value. Nevertheless, the general behaviors observed advocate for $\beta_M^0 = 1100$ as a value that balances randomicity and robusticity in the Metropolis function and allows a good performance of the algorithm, although it could be possible to "fine tune" this value for an increased performance or even to reduce the number of iterations needed for an acceptable result. Another possible modification could be adjusting the β_M^0 value depending on the initial score of the circuit, prior entering the evolution phase.

4.3 Algorithm Results Simulation

The quality of the generated circuit is assessed by the algorithm by means of the Score. However, the degree of reliability of said Score is unknown as it only considers the probability of complex formation by the species conforming the circuit and the absence of an unwanted free toehold. To check that the circuit the algorithm yields is functional, a simulation of the equilibrium states with and without input miRNA was performed through the NUPACK 3.2.2 (Dirks & Pierce, 2003; Dirks & Pierce, 2004; Dirks *et al.*, 2007) suite. The initial concentrations considered for all species was 1 µM. Furthermore, the resulting concentrations in both equilibria were standardized dividing them by 1 µM and multiplying by 100, obtaining this way a percentage of species present in the equilibrium. Species present in a proportion lower than 0.1% were considered absent.

Two circuits were passed through the simulation: the first was produced through mutation plus selection, yielding a score of 97.55, while the second was produced by random mutations without selection by defining $\beta_M^0 = 0$, having a score of $2.97 \cdot 10^{-14}$. The results are shown in ***Figures 11*** & ***12***.

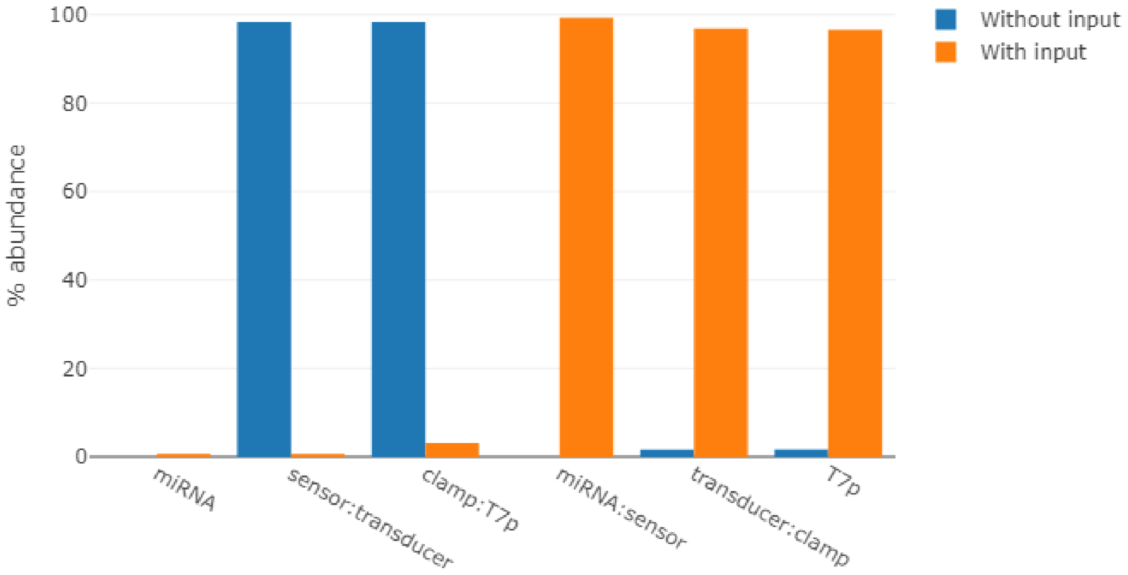


Figure 11: *Equilibrium states simulation comparison of a good scoring circuit*

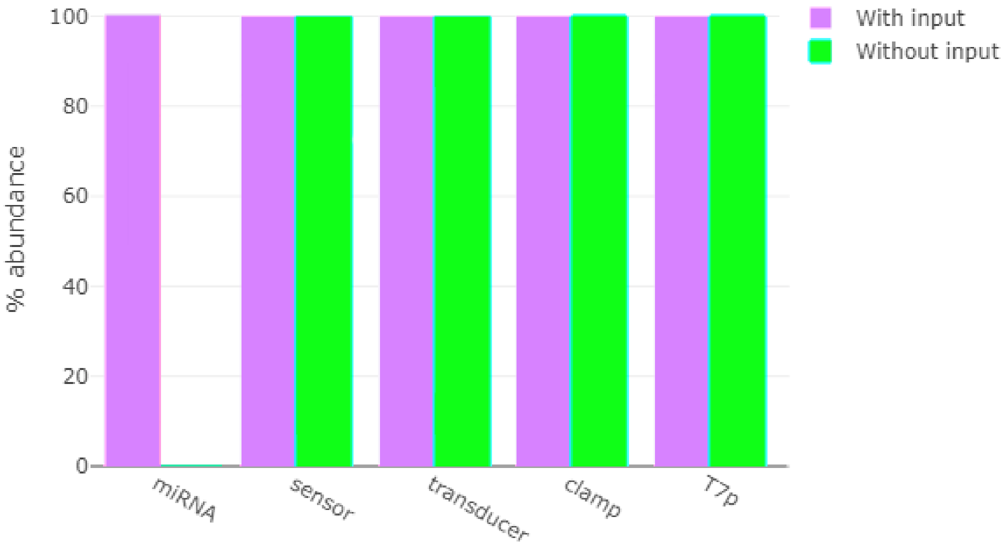


Figure 12: *Equilibrium states simulation comparison of a bad scoring circuit*

As seen in ***Figure 11***, the equilibriums states of the good-scoring circuit are very similar to the ideal equilibrium states shown in ***Figure 5***. In both equilibriums, there is a small amount of noise, when considering T7p in the equilibrium in absence of input miRNA and the complex clamp-T7p in the equilibrium where input miRNA is present. The first one represents background noise while the second represents signal that is not liberated.

Oppositely, in ***Figure 12***, both equilibriums are displayed simultaneously to highlight that, first, there are none of the complexes intended to exist, and second, that there are no differences between states (except the absence of input miRNA). At this point it is safe to conclude that the Score given by the algorithm is truly related to the quality of the circuit generated.

4.4 Kinetic Model Results

4.4.1 Simple Circuit Kinetic Model

Eq. (16) analytically assesses the rate at which T7p is liberated in presence of 1 μM input miRNA and serves as the maximum theoretical rate at which T7p is liberated, since having an amount of 1 μM input miRNA means having all species of the circuit at the same concentration, favoring collisions between them that start the reaction cascade. Nevertheless, to check that the analytical integration has been performed correctly, the rate at which T7p is liberated have been simultaneously estimated through numerical integration (taking 1 μM as initial concentration value for all initial components of the circuit) of the system shown in ***Eq. (9)***, as seen in ***Figure 13***.

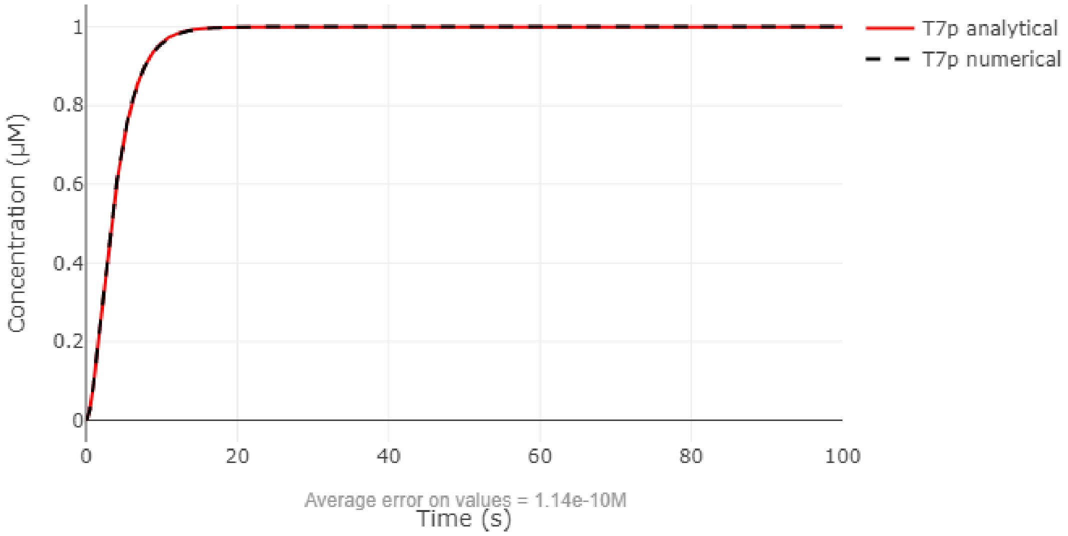


Figure 13*: Comparison between T7p analytical and numerical integration*

The integration seems accurate as the average error on the values is $1.14 \cdot 10^{-10}$ M, which is equivalent to $1.14 \cdot 10^{-4}$ μM. This difference is probably due to the method employed in the numerical integration and Python's memory capacity to keep track of decimal numbers. It can be concluded that the circuit can reach a maximum signal emission (maximum T7p liberation) at around 15 seconds since input miRNA addition.

4.4.2 Signal Amplification Circuit Kinetic Model

The addition of fuel modifies the model as it interacts with species miRNA-sensor, which has a great effect on all other components of the circuit. A numerical integration of ***Eq. (21)*** with initial values 1 μM for all initial components of the circuit yielded the model shown in ***Figure 14***.

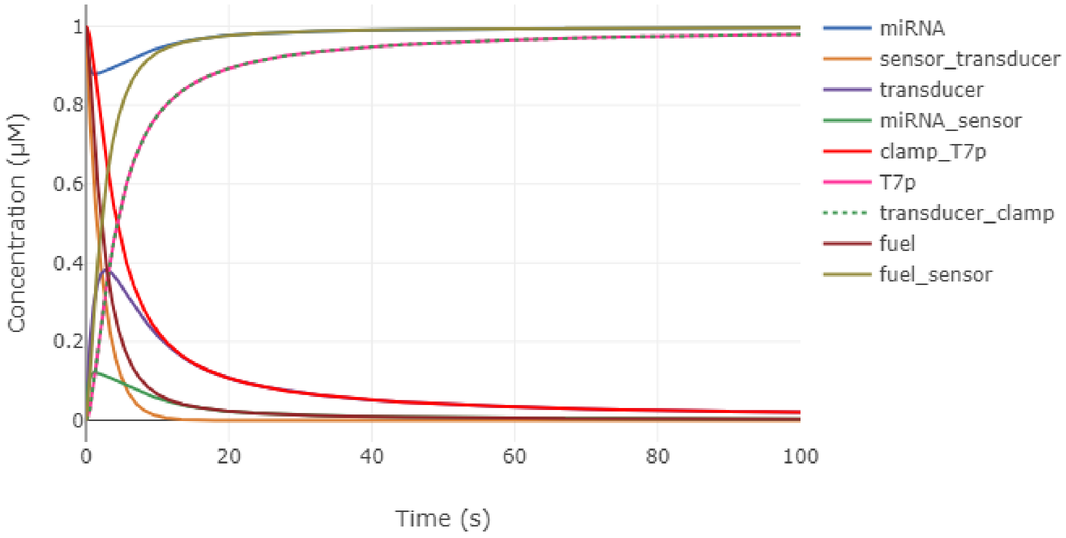


***Figure 14**: Kinetic model for all species participating in the circuit*

The addition of fuel to the circuits slows down the reactions from around 15 seconds to around 100. The cause may be the interference of fuel as the reaction in which fuel displaces miRNA from the miRNA-sensor complex liberates miRNA at much higher rate than miRNA binds to sensor, slowing down the overall circuit. Although the effect of fuel is intended to accelerate the liberation of T7p rather than slowing it down, in the tested conditions (1 µM of input miRNA), it may not be convenient to add this species. Nevertheless, fuel may have a positive effect when signal amplification is actually needed, such as when input miRNA concentrations are very low. In addition, the effects of fuel initial concentration variation are unknown.

4.4.3 Fuel Concentration Effect on Kinetic Model

With the aim of discerning the effect on fuel concentration variation, different fuel concentrations were employed while keeping constant all the other components concentrations (including miRNA). The result is illustrated in ***Figure 15***:

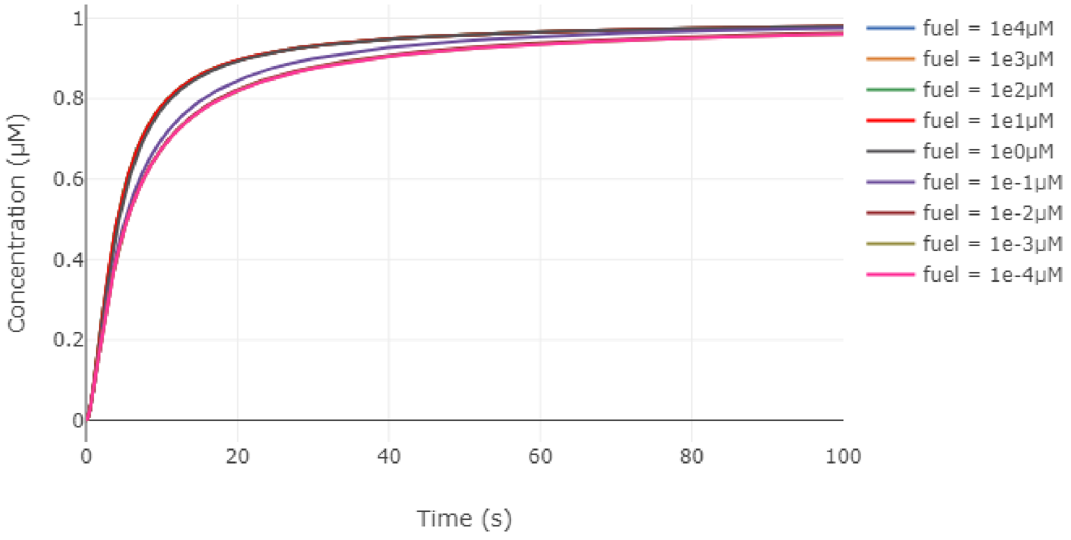


***Figure 15**: T7p concentration evolution at 1 µM input miRNA under the effect of different fuel concentrations*

Fuel concentrations ranging from $1 \cdot 10^{-4}$ µM to 1 µM yield a similar behavior on T7p production. This effect is due to the limiting reagent miRNA-sensor complex in the reaction in which fuel displaces miRNA from said complex. However, when reducing fuel concentration below 1 µM, a drop in the rate of T7p production can be observed. Nevertheless, the effects observed are not dramatic since the miRNA concentration in this experiment was of 1 µM. To observe a more drastic change, miRNA concentration should be reduced. ***Figure 16*** below employs a miRNA concentration of 1 pM:

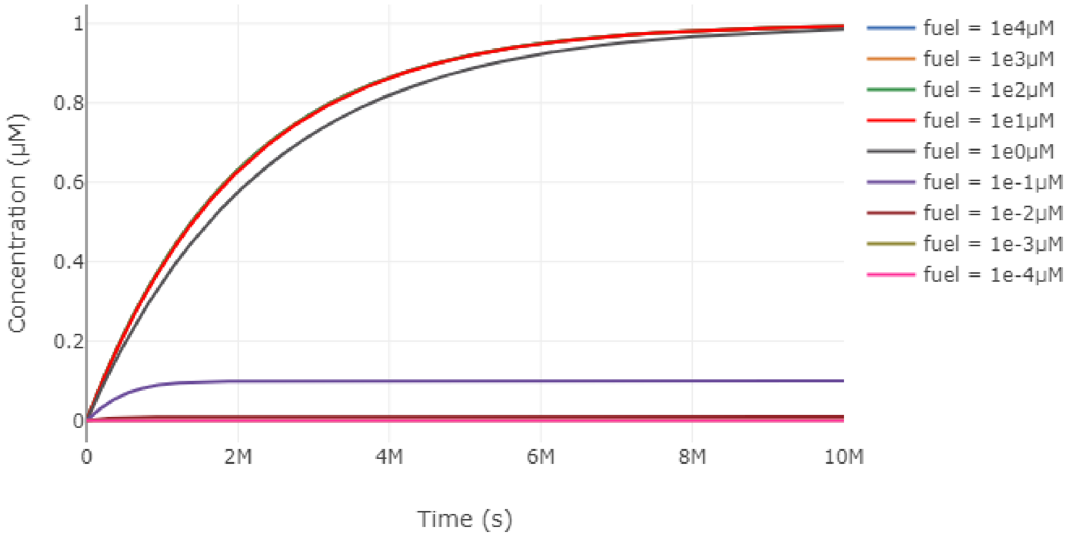


Figure 16*: T7p concentration evolution at 1 pM input miRNA under the effect of different fuel concentrations*

Note that due to the slowness of the reaction, the time axis ranges up to 10 million seconds, which is around 3 months and 26 days. ***Figure 16*** allows a better visualization of the effect of fuel concentration in T7p liberation. There is a sharp drop of reaction speed between fuel at 1 µM and fuel at 0.1 µM and it seems to indicate that the fuel concentration, when miRNA concentration is very low, marks the horizontal asymptote the function is approaching to. To prove this property, the procedure is repeated but with fuel concentrations ranging from 1 µM to 0.1 µM, as seen in ***Figure 17***:

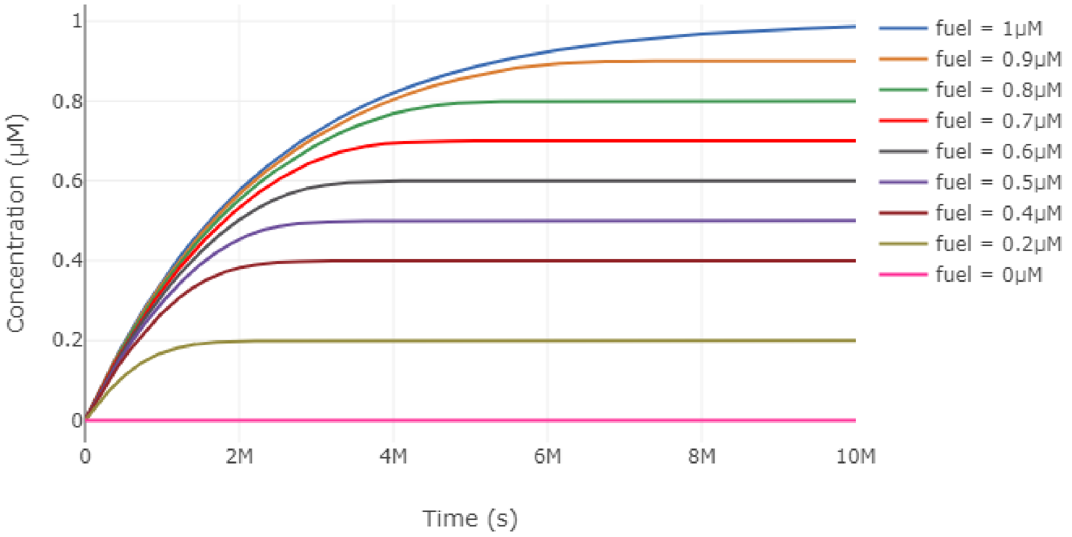


Figure 17*: T7p concentration evolution at 1 pM miRNA under fuel concentrations ranging from 0 to 1 µM*

In ***Figure 17***, when miRNA concentration is low, fuel concentration determines the maximum amount of T7p that the circuit can liberate, as fuel displaces miRNA from miRNA-sensor complex, it allows to reuse this miRNA as fresh input. This cycle is interrupted when all fuel is consumed. It is safe to conclude that the ideal amount of fuel concentration in the circuit is 1 μM as it allows, although after a long time, the liberation of all the T7p in the circuit when triggered by low concentrations of miRNA. If more fuel was added, no significant improvement can be observed, while if adding less, the circuit underperforms.

4.4.4 Concentration of Input miRNA Effect on Reaction Time

In ***Figures 16*** & ***17*** there were hints that lowering miRNA concentration caused reaction time to increase, as it takes more time for a small amount of miRNA to encounter and react with sensor-transducer complex. Up until now in this work, for the sake of simplicity most of the time it was considered that the input miRNA concentration was 1 μM, the same as the other circuit components. The T7p liberation rate with different miRNA concentrations was computed in order to observe the effect on reaction time:

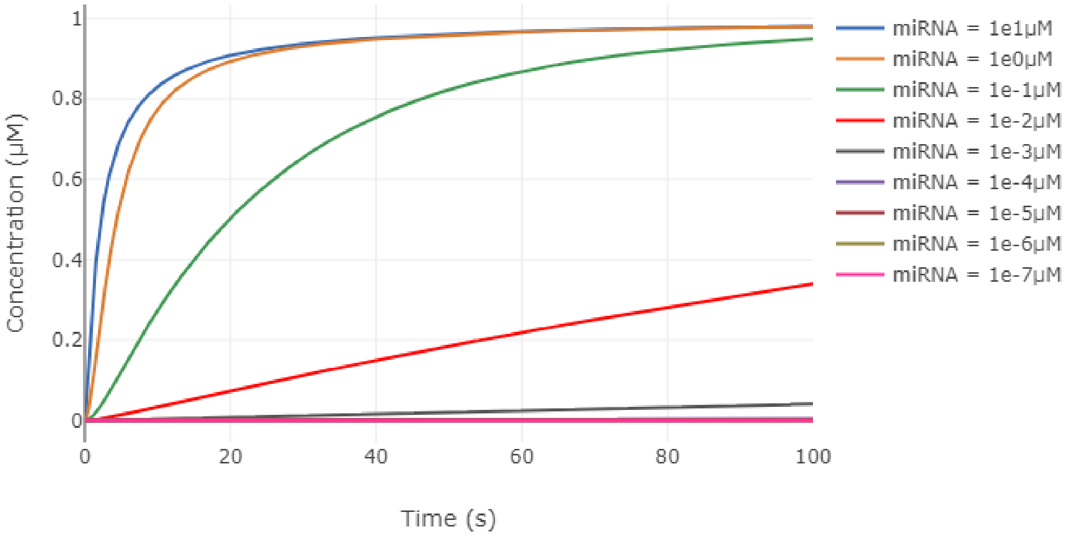


Figure 18: *T7p concentration evolution under the effect of different input miRNA concentrations*

As expected, ***Figure 18*** shows that an increase in 1 order of magnitude (10 μM) of the standard input miRNA concentration barely has any effect on T7p liberation. This effect can be explained through the reaction in which miRNA displaces transducer from sensor-transducer complex, as with excess miRNA but a fixed amount of sensor-transducer, this species turn into the limiting reagent that will govern the remaining reactions taking place in the circuit. However, when reducing 1 order of magnitude instead (0.1 μM), the circuit's performance suffers a sharp drop, which reduces the rate of transducer displacement from complex sensor-transducer by miRNA which, again, will govern the remaining reactions of the circuit. This effect becomes more notable when reducing another order of magnitude (0.01 μM). If reducing input miRNA concentration even further, in an interval of 100 seconds there will be barely any T7p liberated, thus needing more reaction time to be able to detect any signal at all.

To discover the relationship that rules miRNA concentration and reaction time, the time until T7p reached a concentration of 0.95 μM was recorded for different values of miRNA concentration (in μM). The base 10 logarithms of both data pairs were plotted, and a linear regression was performed. Results are shown in ***Figure 19****.*

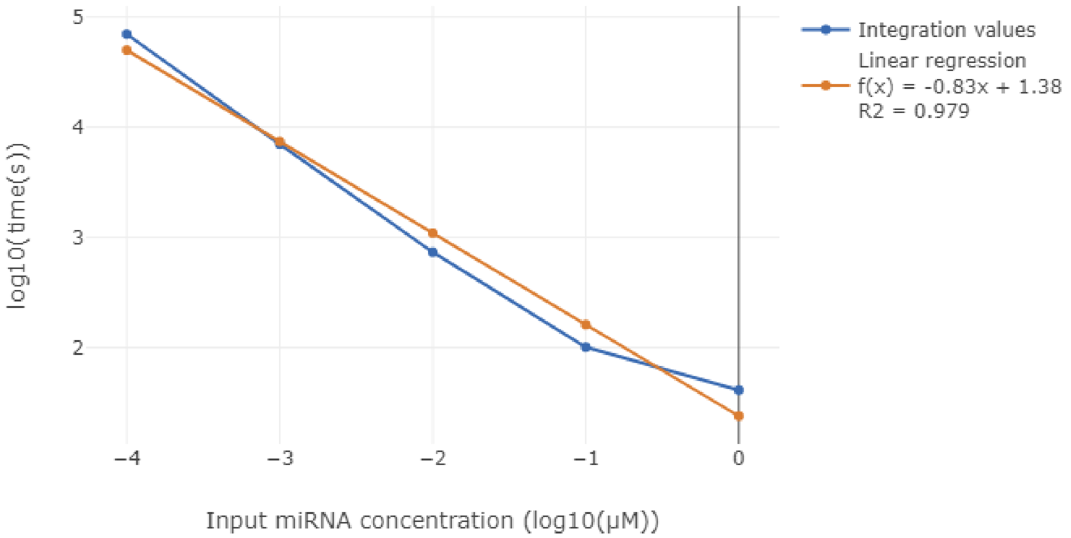


Figure 19*: Relation between the log base 10 of miRNA concentration and log 10 reaction time until 0.95 µM T7p is liberated*

As seen in ***Figure 19*** above, the relationship between base 10 logarithm of reaction time and the base 10 logarithm of miRNA concentration is almost linear, yielding an R^2 value of 0.979. In order to visualize better how this relationship really is, ***Eq. (22)***, given by the linear regression, is solved for reaction time:

$$log_{10}(\tau) = -0.83\, log_{10}[miRNA] + 1.38; \qquad \text{Eq.}\,(22)$$

$$\tau = 10^{log_{10}([miRNA]^{-0.83})} \cdot 10^{1.38};$$

$$\tau = 23.99 \cdot [miRNA]^{-0.83} \qquad \text{Eq.}\,(23)$$

To predict the increase in reaction time due to a decrease of the order of magnitude of miRNA concentration, ***Eq. (23)*** is modified into:

$$\tau = 23.99 \cdot 10^{-0.83x} \qquad \text{Eq.}\,(24)$$

Where:

x: the order of magnitude of miRNA concentration.

Plotting this function yields ***Figure 20***.

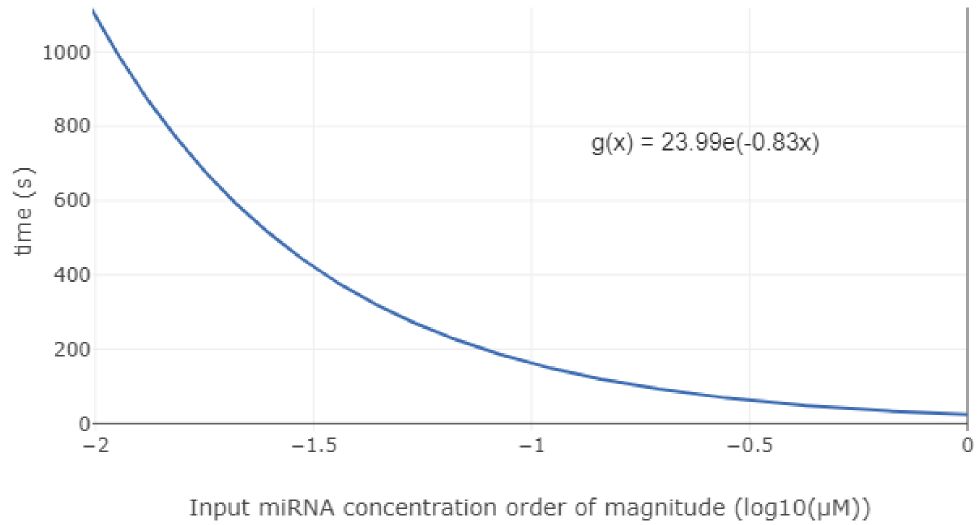


***Figure 20**: Relation between miRNA concentration order of magnitude (in µM) and reaction time until 0.95 µM T7p is liberated*

As seen above, the reaction time increases dramatically with the decrease of miRNA concentration. In addition, ***Eq. (22)*** can establish an approximated detection threshold of the circuit by establishing a maximum reaction time. This maximum is set to 1 week (6.048·10^5 s) as current miRNA detection systems do not take longer periods of time to give a valid result.

$$log_{10}(\tau) = -0.83\, log_{10}[miRNA] + 1.38; \qquad \text{Eq. (22)}$$

$$\frac{1.38 - log_{10}(\tau)}{0.83} = log_{10}[miRNA]\,;$$

$$\frac{1.38 - log_{10}(6.048 \cdot 10^5)}{0.83} = log_{10}[miRNA] = -5.30$$

The threshold is around 10^{-5} µM, which is equivalent to 10 pM. Concentrations below this value will have a reaction time that is much too large for this system to be compelling as a quick miRNA detection system.

4.5 Shadow Circuit Result Simulation

In order to analyze the performance of the Shadow Circuit in signal leak silencing, two equilibria are simulated by means of the NUPACK 3.2.2 suite. The first equilibrium, that represents a state in which the sensor-transducer complex has no signal leak is supposed to have in its equilibrium state the following species: sensor-transducer, S2-TS, AND-AND_clamp. The second equilibrium, that represents a state in which there is maximum leak (in this case due to the low concentration of sensor and S2 which yields high concentrations of free transducer and T2) is supposed to have the following species in its equilibrium: transducer-AND-T2, AND_clamp. The corresponding simulation results are shown in ***Figures 21*** & ***22***.

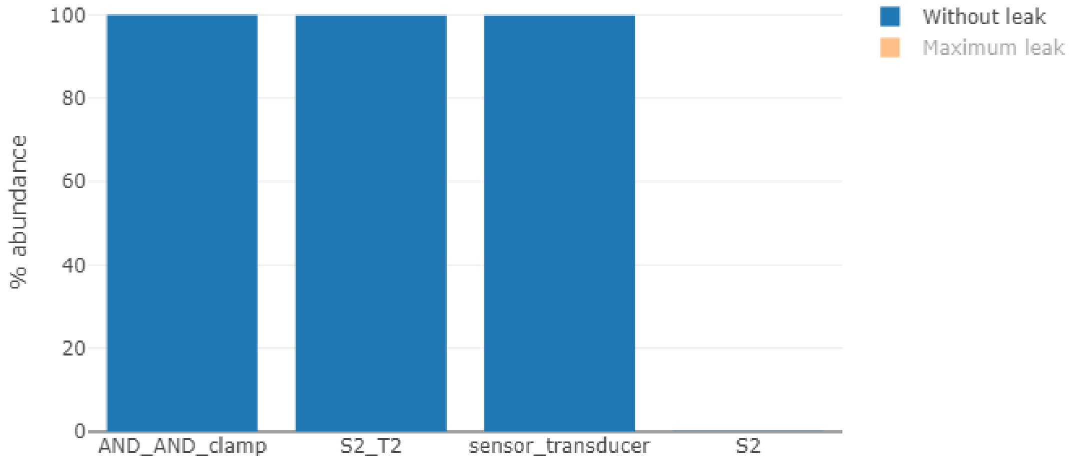


Figure 21*: Species present at the "Without leak" equilibrium*

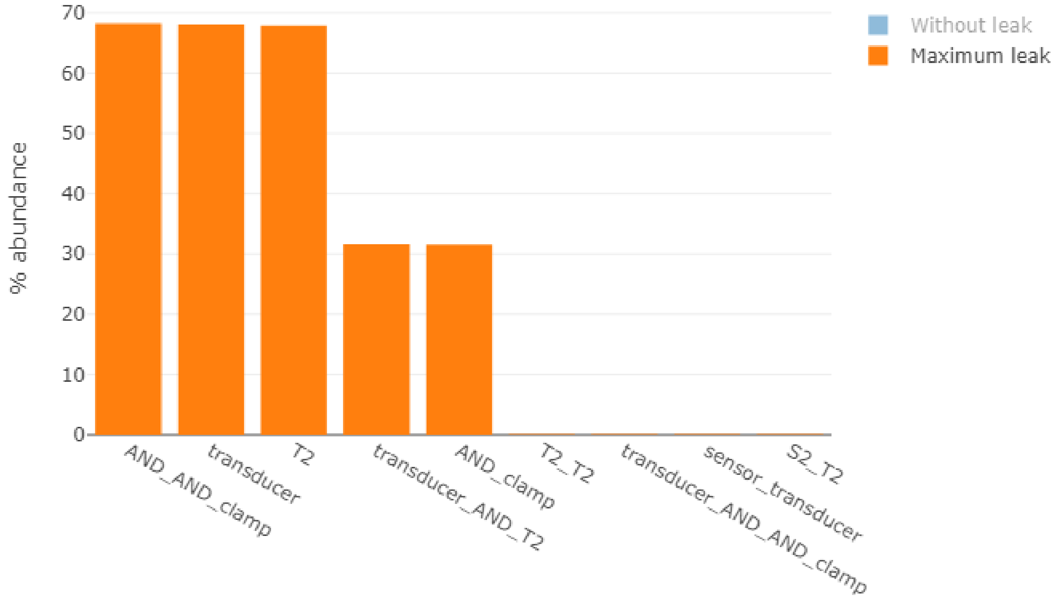


Figure 22*: Species present at the "Maximum leak" equilibrium*

Figure 21 almost shows the ideal species expected to be in the equilibrium where no leak occurs while ***Figure 22*** shows a predominance of species that are only supposed to be at the equilibrium without leak (note that due to the low concentration of sensor and S2, instead of observing species sensor-transducer and S2-T2, free transducer and T2 are observed instead). However, there is a 30% abundance of the complex transducer_AND_T2, indicating that the shadow circuit does kidnap transducer in presence of free T2, assuming that both should come from a leakage with similar kinetics. Nevertheless, the low amount of species transducer_AND_T2 in the equilibrium may indicate that further work should be done in order to optimize the spontaneity of complex formation by means of guided evolution.

5 Conclusions

In the wake of the different results obtained in this work regarding score convergence, randomicity, kinetics and outputs, it is safe to conclude the algorithm is a functional tool that generates viable circuits which could perform in an adequate manner when the input miRNA concentration is higher than 10pM.

The *in silico* design of strand displacement DNA circuits for miRNA detection opens the expectations of a mass production of kits for miRNA detection which may be employed as routine tests in clinic, in underdeveloped countries or even at home, which along with the new discovery of miRNA biomarkers for cancers and neurodegenerative diseases may suppose a turning point in modern diagnostics for these diseases.

However, there are still several limitations to be considered regarding the algorithm developed in this work: the algorithm has a strong dependence of randomicity (at the Metropolis function and the initial sequence generation) which may cause the score to be initially low and to accumulate detrimental mutations that cause the score to not to converge. This means that a single miRNA may cause the algorithm to produce many different good and bad results; output analysis have been performed by the Nupack suite, which is only a simulation tool meaning that experimental testing for the generated circuits may be necessary to improve the algorithm. Another issue related with Nupack is that this tool considers all species to be individual strands when performing the analysis while in reality, most species are pre-hybridized as initial complexes, thus circuits that may not show good results on the Nupack simulation tool could still work properly in reality; the kinetic model is based on theoretical approaches and should be fine-tuned by means of experimental measures.

6 Bibliography

Balcells, I., Cirera, S. & Busk, P.K. (2011). Specific and sensitive quantitative RT-PCR of miRNAs with DNA primers. *BMC Biotechnol,* 11: 70.

Bassi, S. (2010). *Python for bioinformatics*. Boca Raton, FL: CRC Press.

Catalanotto, C., Cogoni, C., & Zardo, G. (2016). MicroRNA in Control of Gene Expression: An Overview of Nuclear Functions. *International Journal Of Molecular Sciences*, *17*(10): 1712.

Chatterjee, A., Leichter, A., Fan, V., Tsai, P., Purcell, R., Sullivan, M., & Eccles, M. (2015). A cross comparison of technologies for the detection of microRNAs in clinical FFPE samples of hepatoblastoma patients. *Scientific Reports*, 5: 10438.

Chen, C., Ridzon, D. A., Broomer, A. J., Zhou, Z., Lee, D. H., Nguyen, J. T., Barbisin, M., Xu, N. L., Mahuvakar, V. R., Andersen, M. R., Lao, K. Q., Livak, K. J., Guegler, K. J. (2005). Real-time quantification of microRNAs by stem-loop RT-PCR. *Nucleic Acids Res,* 33(20): e179.

Dirks, R. M., Bois, J. S., Schaeffer, J. M., Winfree, E., & Pierce, N. A. (2007). Thermodynamic analysis of interacting nucleic acid strands. *SIAM Rev*, 49:65-88.

Dirks, R. M. & Pierce, N. A. (2004). An algorithm for computing nucleic acid base-pairing probabilities including pseudoknots. *J Comput Chem*, 25:1295-1304.

Dirks, R. M. & Pierce, N. A. (2003). A partition function algorithm for nucleic acid secondary structure including pseudoknots. *J Comput Chem*, 24:1664-1677.

Draghici, S., Khatri, P., Eklund, A., & Szallasi, Z. (2006). Reliability and reproducibility issues in DNA microarray measurements. *Trends Genetics*, 22(2): 101–9.

Ekmekci, B., McAnany, C., & Mura, C. (2016). An Introduction to Programming for Bioscientists: A Python-Based Primer. *PLOS Computational Biology*, *12*(6), e1004867

Heron, M.P. (2018) Deaths: Leading Causes for 2016. *National Vital Statistics Reports; Centers for Disease Control and Prevention: Atlanta, GA, USA*; 67

Kumar, P., Dezso, Z., MacKenzie, C., Oestreicher, J., Agoulnik, S., Byrne, M., Bernier, F., Yanagimachi, M., Aoshima, K. & Oda, Y. (2013). Circulating miRNA Biomarkers for Alzheimer's Disease. *PLoS ONE*, 8(7): e69807.

Li, J., Batcha, A., Gaining, B., & Mansmann, U. (2015). An NGS workflow blueprint for DNA sequencing data and its application in individualized molecular oncology. *Cancer Informatics*, 14(Suppl 5): 87–107.

Lorenz, R., Bernhart, S., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P., & Hofacker, I. (2011). ViennaRNA Package 2.0. *Algorithms For Molecular Biology*, 6(1): 26.

Moody, L., He, H., Pan, Y., & Chen, H. (2017). Methods and novel technology for microRNA quantification in colorectal cancer screening. *Clinical Epigenetics*, *9*(1): 119.

PLOTLY. Viewed on May 1st, 2019. Available at: https://plot.ly

Pockar, S., Globocnik Petrovic, M., Peterlin, B., & Vidovic Valentincic, N. (2019). MiRNA as biomarker for uveitis - A systematic review of the literature. *Gene*, *696*, 162-175.

PYTHON SOFTWARE FOUNDATION. Viewed on May 1st, 2019. Available at: https://www.python.org/

Qiu, L., Tan, E., & Zeng, L. (2015). microRNAs and Neurodegenerative Diseases. *Advances In Experimental Medicine And Biology*, 85-105.

Redshaw, N., Wilkes, T., Whale, A., Cowen, S., Huggett, J., & Foy, C. (2013). A comparison of miRNA isolation and RT-qPCR technologies and their effects on quantification accuracy and repeatability. *BioTechniques,* 54(3): 155–64.

Salis, H., Mirsky, E., & Voigt, C. (2009). Automated design of synthetic ribosome binding sites to control protein expression. *Nature Biotechnology*, *27*(10), 946-950.

SCIPY. Viewed on June 3rd, 2019. Available at: https://www.scipy.org/

Seelig, G., Soloveichik, D., Zhang, D., & Winfree, E. (2006). Enzyme-Free Nucleic Acid Logic Circuits. *Science*, *314*(5805), 1585-1588.

Song, T., Gopalkrishnan, N., Eshra, A., Garg, S., Mokhtar, R., & Bui, H. et al. (2018). Improving the Performance of DNA Strand Displacement Circuits by Shadow Cancellation. *ACS Nano*, *12*(11), 11689-11697.

Soriano, J., Rojas-Rueda, D., Alonso, J., Antó, J., Cardona, P., Fernández, E., Garcia-Basteiro, A., Benavides, F., Glenn, S., Krish, V., Lazarus, J., Martínez-Raga, J., Masana, M., Nieuwenhuijsen, M., Ortiz, A., Sánchez-Niño, M., Serrano-Blanco, A., Tortajada-Girbés, M., Tyrovolas, S., Haro, J., Naghavi, M. & Murray, C. (2018). The burden of disease in Spain: Results from the Global Burden of Disease 2016. *Medicina Clínica (English Edition)*, 151(5): 171-190.

Srinivas, N., Ouldridge, T., Šulc, P., Schaeffer, J., Yurke, B., & Louis, A. et al. (2013). On the biophysics and kinetics of toehold-mediated DNA strand displacement. *Nucleic Acids Research*, *41*(22), 10641-10658.

Wahid, F., Shehzad, A., Khan, T., & Kim, Y. (2010). MicroRNAs: Synthesis, mechanism, function, and recent clinical trials. *Biochimica Et Biophysica Acta (BBA) - Molecular Cell Research*, *1803*(11), 1231-1243.

Wang, B., Thachuk, C., Ellington, A., Winfree, E., & Soloveichik, D. (2018). Effective design principles for leakless strand displacement systems. *Proceedings Of The National Academy Of Sciences*, *115*(52), e12182-e12191.

Wang, B., & Xi, Y. (2013). Challenges for microRNA microarray data analysis. *Microarrays (Basel)*, 2(2): 34–50.

Wang, K., Yuan, Y., Cho, J., McClarty, S., Baxter, D., & Galas, D. (2012). Comparing the MicroRNA Spectrum between Serum and Plasma. *Plos ONE*, *7*(7): e41561.

Wittmann, J. & Jäck, H. (2010). Serum microRNAs as powerful cancer biomarkers. *Biochimica et Biophysica Acta (BBA) - Reviews on Cancer*, 1806(2): 200-207.

Wu, D., Hu, Y., Tong, S., Williams, B., Smyth, G., & Gantier, M. (2013). The use of miRNA microarrays for the analysis of cancer samples with global miRNA decrease. *RNA*, 19(7): 876–88.

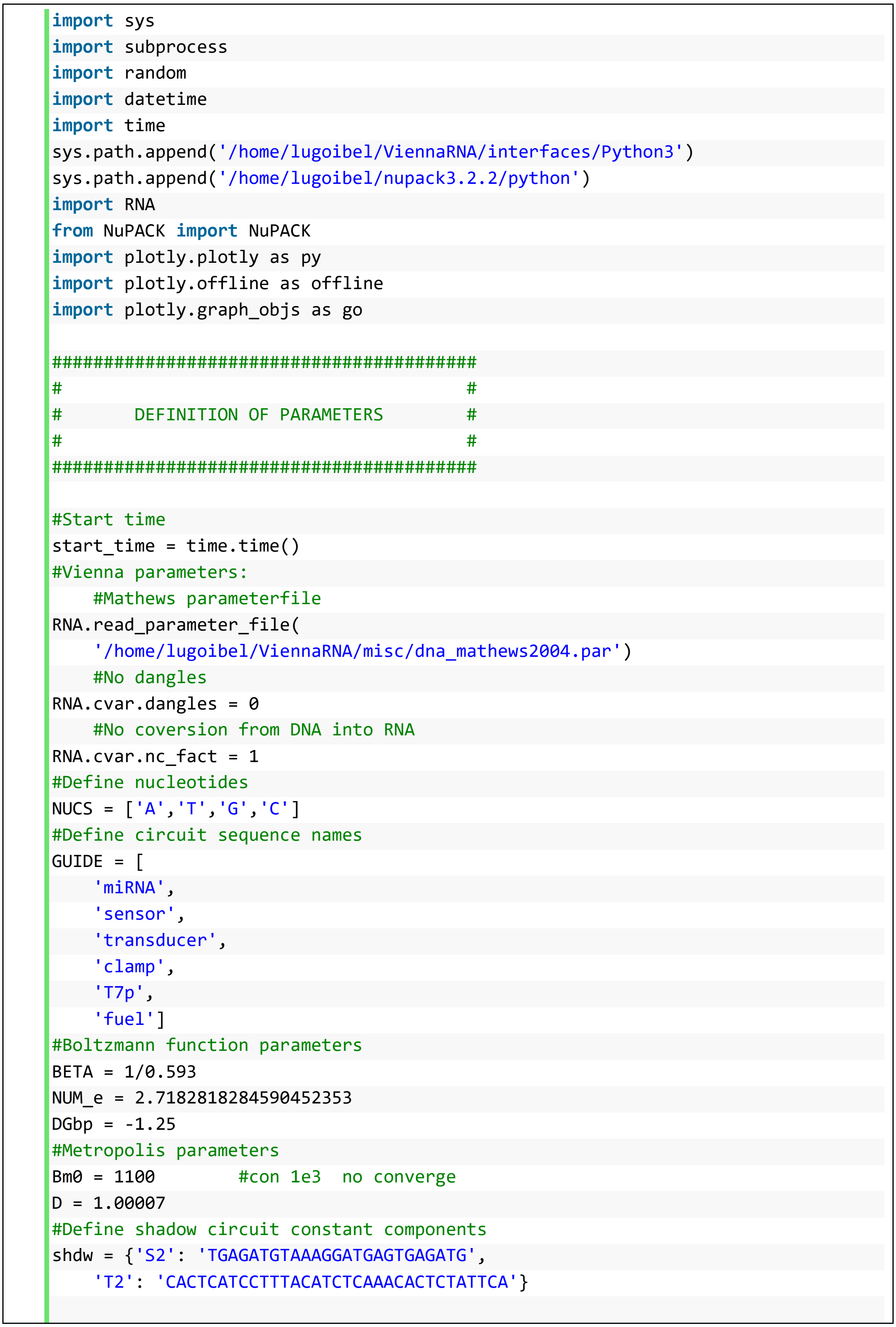
Zhang, D., Turberfield, A., Yurke, B., & Winfree, E. (2007). Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA. *Science*, *318*(5853), 1121-1125.

Zhang, D., & Seelig, G. (2011). Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry*, *3*(2), 103-113.

Zhang, D., & Winfree, E. (2009). Control of DNA Strand Displacement Kinetics Using Toehold Exchange. *Journal Of The American Chemical Society*, *131*(47), 17303-17314.

7 Annex I: Python code of the developed algorithm.

Note that “'/home/lugoibel/ViennaRNA/interfaces/Python3'” and “'/home/lugoibel/nupack3.2.2/python'” are the absolute paths of the ViennaRNA python library and the Nupack wrapper (Salis *et al.*, 2009) (**Annex II**) employed in this work.



```python
import sys
import subprocess
import random
import datetime
import time
sys.path.append('/home/lugoibel/ViennaRNA/interfaces/Python3')
sys.path.append('/home/lugoibel/nupack3.2.2/python')
import RNA
from NuPACK import NuPACK
import plotly.plotly as py
import plotly.offline as offline
import plotly.graph_objs as go

#########################################
#                                       #
#       DEFINITION OF PARAMETERS        #
#                                       #
#########################################

#Start time
start_time = time.time()
#Vienna parameters:
    #Mathews parameterfile
RNA.read_parameter_file(
    '/home/lugoibel/ViennaRNA/misc/dna_mathews2004.par')
    #No dangles
RNA.cvar.dangles = 0
    #No coversion from DNA into RNA
RNA.cvar.nc_fact = 1
#Define nucleotides
NUCS = ['A','T','G','C']
#Define circuit sequence names
GUIDE = [
    'miRNA',
    'sensor',
    'transducer',
    'clamp',
    'T7p',
    'fuel']
#Boltzmann function parameters
BETA = 1/0.593
NUM_e = 2.7182818284590452353
DGbp = -1.25
#Metropolis parameters
Bm0 = 1100        #con 1e3  no converge
D = 1.00007
#Define shadow circuit constant components
shdw = {'S2': 'TGAGATGTAAAGGATGAGTGAGATG',
    'T2': 'CACTCATCCTTTACATCTCAAACACTCTATTCA'}

```

Annex I: *Python code of the developed algorithm (continues on the next page)*

```python
#########################################
#                                       #
#       DEFINITION OF FUNCTIONS         #
#                                       #
#########################################

#Define command line input system
def cmdinput():
    global USERINPUT
    global GUIDE
    looping = True
    while looping:
        if 'U' in USERINPUT:
            USERINPUT = USERINPUT.replace(
                'U','T')
        UNIQ = set(USERINPUT)
        #Checks if input is a sequence of adequate length
        if (UNIQ.issubset(NUCS) and
                len(USERINPUT) >= 20):
            seqs_preit[GUIDE[0]] = USERINPUT[:25]
            looping = False
        #Checks if input is meant to be a test
        elif USERINPUT == 'TEST':
            GUIDE = ['Rodrigo_miRNA'] + GUIDE[1:]
            seqs_preit['Rodrigo_miRNA'] = 'TGGAGTGTGACAATGGTGTTTG'
            looping = False
        #Exit system
        elif USERINPUT == 'EXIT':
            exit()
        #Retry input if previous statements are false
        else:
            USERINPUT = input(
                'Enter a VALID input: '
                ).upper()
    return None

#Define fasta file input system. Saves data in a dictionary as
#key = header and value = sequence, only if the sequence is
#adequate
def fileinput():
    dict = {}

    for line in open(USERINPUT):
        line = line.strip('\n')

        if line[0] == '>':
            key = line[1:].split()[0]
            value = ''

        else:
            value += line

        if (set(value).issubset(NUCS) and
                len(value) >= 20):
            dict[key] = value[:25]
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
    return dict

#Define reverse complementary generator
def revcomp(seq):
    seq = seq.upper(
        ).replace('A','t'
        ).replace('T','a'
        ).replace('G','c'
        ).replace('C','g'
        )[::-1].upper()
    return seq

#Random sequence builder
def randseq(length):
    out = ''
    for n in range(length):
        out += random.sample(NUCS, 1)[0]
    return out

#Define circuit core sequences generator
def genseq(miRNA, prom):
    n = len(miRNA)
    rootseq = (miRNA.upper()
        + randseq(5)#'TATTC'
        + prom)
    sensor = revcomp(
        rootseq[: n+8])
    transducer = rootseq[6:]
    clamp = revcomp(
        rootseq[n - 1 :])
    fuel = rootseq[6: n + 8]
    return (sensor,
        transducer,
        clamp,
        fuel)

#Define Boltzmann function
def bolfunc(seq1, seq2, seq_DG):
    Pairkey = (seq1
        + '_'
        + seq2)

    Numerator = NUM_e**(- BETA*seq_DG[Pairkey])
    Denominator = Numerator

    if seq1 == GUIDE[0]:
        SecondKey = 'sensor_transducer'

    elif seq1 == 'transducer':
        SecondKey = 'clamp_T7p'

    elif seq1 == 'fuel':
        SecondKey = GUIDE[0] + '_sensor'

    Denominator += NUM_e**(- BETA*seq_DG[SecondKey])
```

Annex I: *Python code of the developed algorithm (continues on the next page)*

```python
    func = Numerator/Denominator

    return func

#Define function for probability calculation employed in
#secondary pairments
def probfunc(seq1, seq2, seq_DG, seqs):
    Pairkey = (seq1
        + '_'
        + seq2)
    Numerator = NUM_e**(- BETA*seq_DG[Pairkey])

    if seq1 == 'sensor':
        L = 19

    if seq1 == 'clamp':
        L = len(seqs[GUIDE[4]])
    Denominator = NUM_e**(- BETA*L*DGbp)
    func = Numerator/Denominator

    if func > 1:
        func = 1
    return func

#Define toehold score function
def toeholdscore(name, seq_ss):
    DIST = (len(seqs_preit['transducer'])
        - len(seqs_preit['T7p'])
        + 3)
    struct = seq_ss[name].split(
        '&'
        )[1][(DIST-6):DIST]
    j = 0

    for symbol in struct:
        if symbol == '.':
            j += 1
    return j

#Define Packing and Scoring function.
def scorefunc(seqs):
    seq_DG = {}
    seq_ss = {}
    i = -1
    #Saves in a dictionary the MFE and structure of circuit pairs
    for seq1 in GUIDE[:-2]:
        i += 1
        seq2 = GUIDE[i + 1]
        name = (seq1
            + '_'
            + seq2)
        (ss, mfe) = RNA.cofold(
            (seqs[seq1]
            + '&'
            + seqs[seq2]))
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
        seq_DG[name] = mfe
        seq_ss[name] = (ss[: len(seqs[seq1])]
            + '&'
            + ss[(len(seqs[seq1])) :-1])

    (ss, mfe) = RNA.cofold(
        (seqs['fuel']
        + '&'
        + seqs['sensor']))
    seq_DG['fuel_sensor'] = mfe
    seq_ss['fuel_sensor'] = (ss[: len(seqs['fuel'])]
        + '&'
        + ss[(len(seqs['fuel'])) :-1])
    #Caulculates pair probabilities and Score

    P1 = bolfunc(
        GUIDE[0],
        'sensor',
        seq_DG)
    P2 = bolfunc(
        'transducer',
        'clamp',
        seq_DG)
    P3 = probfunc(
        'sensor',
        'transducer',
        seq_DG,
        seqs)
    P4 = probfunc(
        'clamp',
        'T7p',
        seq_DG,
        seqs)
    P5 = bolfunc(
        'fuel',
        'sensor',
        seq_DG)
    T = toeholdscore('sensor_transducer', seq_ss)

    score = P1*P2*P3*P4*P5*(6-T)/6
    dats = [P1,P2,P3,P4,P5,T,score]
    return dats

#Define mutation function
def mutf(seqs):
    seqs_aftermutation = {}

    #Creates a new dictionary with sequences
    for element in seqs:
        seqs_aftermutation[element] = seqs[element]

    #Creates a new guidelist excluding miRNA and T7p
    mutlist = GUIDE[1:-2] + [GUIDE[-1]]

    #Chooses a random base from a random sequence from ensemble
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```python
    target_name = random.sample(mutlist, 1)[0]
    target_seq = list(seqs[target_name])
    position = random.randint(0, (len(target_seq) - 1))
    base = random.sample(NUCS, 1)[0]

    while base == target_seq[position]:
        base = random.sample(NUCS, 1)[0]

    #Writes the mutated sequence
    target_seq[position] = base
    target_seq = ''.join(target_seq)
    seqs_aftermutation[target_name] = target_seq

    return seqs_aftermutation

#Define a function that interprets NuPACK output files
def eqcon(dict, guide):
    outlist = []
    outdict = {}

    for el in dict['complexes_concentrations']:
        stand = round((float(el[-1])/1e-8), 2)

        if stand < 0.1:
            continue

        cmplx = list(map(int, el[0:-2]))

        name = []
        i = -1
        for n in cmplx:
            i += 1

            if n:
                name += n*[guide[i]]

        name = '_'.join(name)

        outlist += [name]
        outdict[name] = [el[-1], stand]
    return outlist, outdict

#Define test-tube prediction of final equilibriums by means of NuPACK
def test_tube(seqs, guide):
    print('Calculating test-tube NuPACK simulation')
    seq_list = []
    concent = [1e-6, 1e-6]

    if 'fuel' not in guide:
        concent += [1e-6, 1e-6]

    for el in guide:
        seq_list += [seqs[el]]

    eq_1 = NuPACK(
```

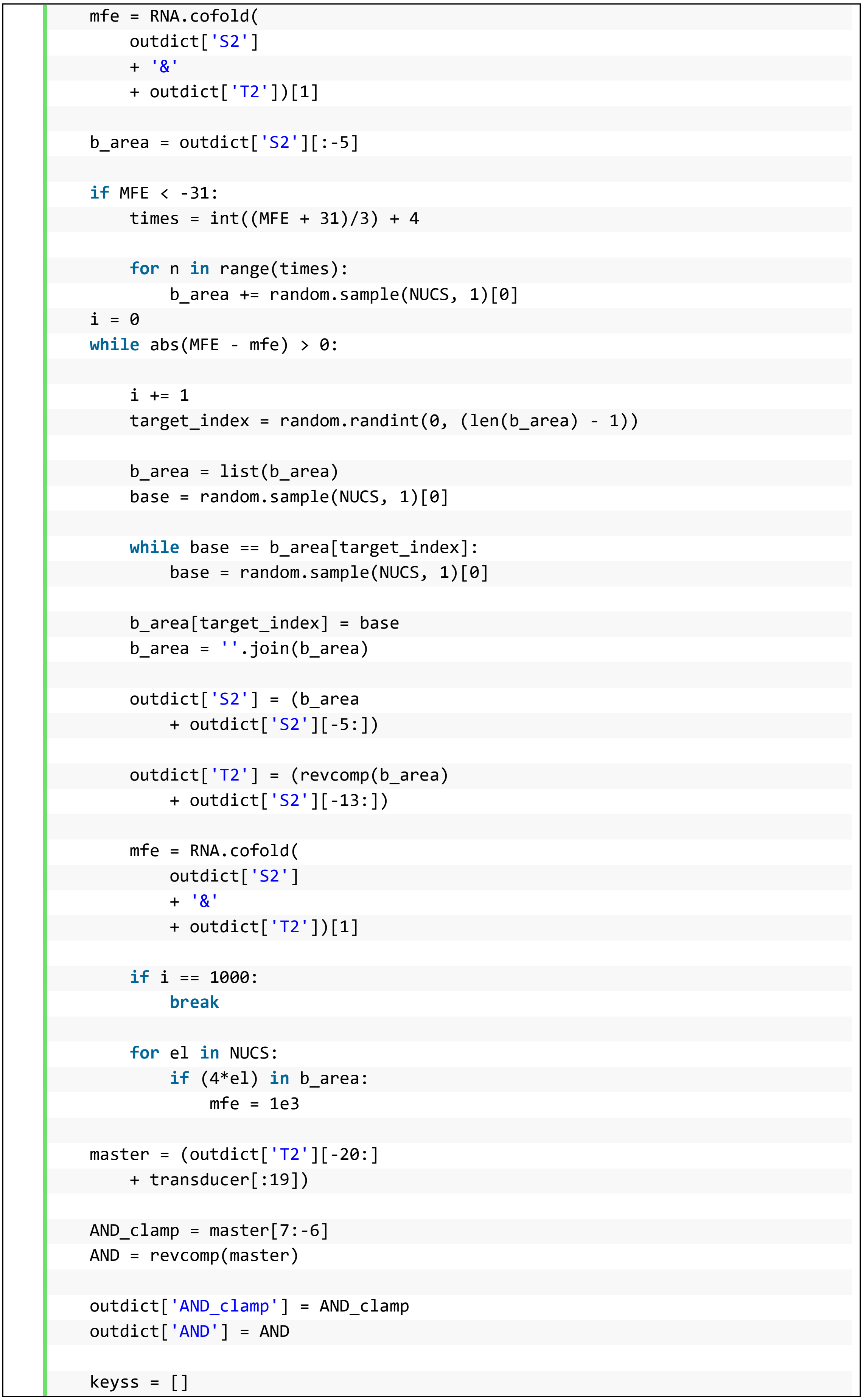
***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
        Sequence_List=seq_list,
        material='dna')
    eq_2 = NuPACK(
        Sequence_List=seq_list,
        material='dna')

    eq_1.complexes(
        dangles='none',
        MaxStrands=2,
        quiet=True)
    eq_2.complexes(
        dangles='none',
        MaxStrands=2,
        quiet=True)

    eq_1.concentrations(
        concentrations=[1e-6] + concent,
        quiet=True)
    eq_2.concentrations(
        concentrations=[1e-9] + concent,
        quiet=True)

    (eq_1order, eq_1) = eqcon(eq_1, guide)
    (eq_2order, eq_2) = eqcon(eq_2, guide)
    EQUILIBRIUMGUIDES = [eq_1order, eq_2order]
    return EQUILIBRIUMGUIDES, eq_1, eq_2

#Define bar-chart plot function for NuPACK test-tube prediction
def eqsbarplot(guides, dict1, dict2):
    global timessufix
    dat1 = []
    dat2 = []

    for list in guides:
        for el in list:

            if guides[0] == list:
                dat1 += [dict1[el][-1]]

            else:
                dat2 += [dict2[el][-1]]

    trace1 = go.Bar(
        x=guides[0],
        y=dat1,
        name='With input')
    trace2 = go.Bar(
        x=guides[1],
        y=dat2,
        name='Without input')

    data = [trace1, trace2]
    layout = go.Layout(
        barmode='group',
        title='Equilibrium concentrations for species',
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```python
        yaxis=dict(title='% abundance'))
    fig = go.Figure(
        data=data,
        layout=layout)
    filename = ('Equilibrium_study_'
        + timesuffix
        + '.html')
    offline.plot(
        fig,
        filename=filename,
        auto_open=False)

    return None

#Define metropolis function to induce random sampling
def Metropolis():
    global Dats_preit, Score_preit, seqs_preit
    Bmk = Bm0*(D**k)
    M = NUM_e**(
        - Bmk*(
            Score_preit
            - Score_posit))

    if random.random() < M:
#        print('\nMetropolis MUTATED\n')
        Dats_preit = Dats_posit
        Score_preit = Score_posit
        seqs_preit = seqs_posit
    return None

#Define percentage progress percentage function
def progress():
    global perc_0
    perc_1 = (k/100000)*100

    if int(perc_1/5) > int(perc_0/5):
        perc_0 = perc_1
        print(
            'Status: '
            + str(int(perc_0))
            + '% completed')
    return None

#Define shadow circuit generation function
def shadowcirc(transducer):
    outdict = {}

    for el in shdw:
        outdict[el] = shdw[el]

    MFE = RNA.cofold(
        seqs_preit['sensor']
        + '&'
        + seqs_preit['transducer'])[1]
```

Annex I*: Python code of the developed algorithm (continues on the next page)*



```
    mfe = RNA.cofold(
        outdict['S2']
        + '&'
        + outdict['T2'])[1]

    b_area = outdict['S2'][:-5]

    if MFE < -31:
        times = int((MFE + 31)/3) + 4

        for n in range(times):
            b_area += random.sample(NUCS, 1)[0]
    i = 0
    while abs(MFE - mfe) > 0:

        i += 1
        target_index = random.randint(0, (len(b_area) - 1))

        b_area = list(b_area)
        base = random.sample(NUCS, 1)[0]

        while base == b_area[target_index]:
            base = random.sample(NUCS, 1)[0]

        b_area[target_index] = base
        b_area = ''.join(b_area)

        outdict['S2'] = (b_area
            + outdict['S2'][-5:])

        outdict['T2'] = (revcomp(b_area)
            + outdict['S2'][-13:])

        mfe = RNA.cofold(
            outdict['S2']
            + '&'
            + outdict['T2'])[1]

        if i == 1000:
            break

        for el in NUCS:
            if (4*el) in b_area:
                mfe = 1e3

    master = (outdict['T2'][-20:]
        + transducer[:19])

    AND_clamp = master[7:-6]
    AND = revcomp(master)

    outdict['AND_clamp'] = AND_clamp
    outdict['AND'] = AND

    keyss = []
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
    for el in outdict.keys():
        keyss += [el]
    keyss.sort()

    return outdict, keyss

#MAIN
def main():
    global k, timesuffix, perc_0, seqs_preit, seqs_posit
    global Score_preit, Score_posit, Dats_preit, Dats_posit

    #Moment in time:
    timesuffix = '_'.join(
        str(datetime.datetime.now()
        ).split())

    (seqs_preit['sensor'],
    seqs_preit['transducer'],
    seqs_preit['clamp'],
    seqs_preit['fuel']) = genseq(seqs_preit[GUIDE[0]], seqs_preit['T7p'])

    Dats_preit = scorefunc(seqs_preit)
    Score_preit = Dats_preit[-1]

    (equilibriumguide,
    eq_1,
    eq_2) = test_tube(seqs_preit, GUIDE[:-1])

    fuelguide = GUIDE[:2] + [GUIDE[-1]]
    fuelguide = fuelguide[::-1]

    (equilibriumguide_fuel,
    w_fuel,
    wo_fuel) = test_tube(seqs_preit, fuelguide)

    OUTFILE = open(
        'Output_'
        + GUIDE[0]
        + '_'
        + timesuffix
        + '.txt',
        'w')
    OUTFILE.write('This is the output of your job done on '
        + timesuffix
        + '\n')

    for el in GUIDE:
        OUTFILE.write('>'
            + el
            + '\n'
            + seqs_preit[el]
            + '\n')

    OUTFILE.write('\nP1 = ' + str(Dats_preit[0]) + '\n')
    OUTFILE.write('P2 = ' + str(Dats_preit[1]) + '\n')
```
***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
    OUTFILE.write('P3 = ' + str(Dats_preit[2]) + '\n')
    OUTFILE.write('P4 = ' + str(Dats_preit[3]) + '\n')
    OUTFILE.write('P5 = ' + str(Dats_preit[4]) + '\n')
    OUTFILE.write('Toehold = ' + str(Dats_preit[5]) + '\n')
    OUTFILE.write('Score = ' + str(Score_preit) + '\n')
    OUTFILE.write('Standarized score = '
        + str(Score_preit*100/Dats_preit[3])
        + '\n')


    OUTFILE.write('\n------WITH INPUT------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide[0]:
        OUTFILE.write(el
            + '\t'
            + eq_1[el][0]
            + '\t'
            + str(eq_1[el][1])
            + '\n')

    OUTFILE.write('\n------WITHOUT INPUT------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide[1]:
        OUTFILE.write(el
            + '\t'
            + eq_2[el][0]
            + '\t'
            + str(eq_2[el][1])
            + '\n')

    OUTFILE.write('\nFuel transduction assessment\n')
    OUTFILE.write('\n------WITH FUEL------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide_fuel[0]:
        OUTFILE.write(el
            + '\t'
            + w_fuel[el][0]
            + '\t'
            + str(w_fuel[el][1])
            + '\n')

    OUTFILE.write('\n------WITHOUT FUEL------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```python
    for el in equilibriumguide_fuel[1]:
        OUTFILE.write(el
            + '\t'
            + wo_fuel[el][0]
            + '\t'
            + str(wo_fuel[el][1])
            + '\n')
    OUTFILE.write('\n')

    #1e5 cycles of mutations and selection following the global score

    k = 0
    perc_0 = 0

    for n in range(int(1e5)):
        k += 1
        seqs_posit = mutf(seqs_preit)
        Dats_posit = scorefunc(seqs_posit)
        Score_posit = Dats_posit[-1]

        if Score_posit >= Score_preit:
            Dats_preit = Dats_posit
            Score_preit = Score_posit
            seqs_preit = seqs_posit

        else:
            Metropolis()

        progress()

    (equilibriumguide,
    eq_1,
    eq_2) = test_tube(seqs_preit, GUIDE[:-1])
    eqsbarplot(equilibriumguide,
        eq_1,
        eq_2)

    (equilibriumguide_fuel,
    w_fuel,
    wo_fuel) = test_tube(seqs_preit, fuelguide)
    #OUTFILE = open('Output_'+GUIDE[0]+timesuffix+'.txt', 'w')
    #OUTFILE.write('This is the output of your job done on '+timesuffix+'\n')

    for el in GUIDE:
        OUTFILE.write('>'
            + el
            + '\n'
            + seqs_preit[el]
            + '\n')

    OUTFILE.write('\nP1 = ' + str(Dats_preit[0]) + '\n')
    OUTFILE.write('P2 = ' + str(Dats_preit[1]) + '\n')
    OUTFILE.write('P3 = ' + str(Dats_preit[2]) + '\n')
    OUTFILE.write('P4 = ' + str(Dats_preit[3]) + '\n')
    OUTFILE.write('P5 = ' + str(Dats_preit[4]) + '\n')
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
    OUTFILE.write('Toehold = ' + str(Dats_preit[5]) + '\n')
    OUTFILE.write('Score = ' + str(Score_preit) + '\n')
    OUTFILE.write('Standarized score = '
        + str(Score_preit*100/Dats_preit[3])
        + '\n')

    OUTFILE.write('\n------WITH INPUT------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide[0]:
        OUTFILE.write(el
            + '\t'
            + eq_1[el][0]
            + '\t'
            + str(eq_1[el][1])
            + '\n')

    OUTFILE.write('\n------WITHOUT INPUT------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide[1]:
        OUTFILE.write(el
            + '\t'
            + eq_2[el][0]
            + '\t'
            + str(eq_2[el][1])
            + '\n')

    OUTFILE.write('\nFuel transduction assessment\n')
    OUTFILE.write('\n------WITH FUEL------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide_fuel[0]:
        OUTFILE.write(el
            + '\t'
            + w_fuel[el][0]
            + '\t'
            + str(w_fuel[el][1])
            + '\n')

    OUTFILE.write('\n------WITHOUT FUEL------')
    OUTFILE.write('\nComplexes')
    OUTFILE.write('\tConcentration (M)')
    OUTFILE.write('\tStandarized (%)\n')

    for el in equilibriumguide_fuel[1]:
        OUTFILE.write(el
            + '\t'
            + wo_fuel[el][0]
```

***Annex I**: Python code of the developed algorithm (continues on the next page)*

```
            + '\t'
            + str(wo_fuel[el][1])
            + '\n')


    (shadow, shadowguide) = shadowcirc(
        seqs_preit['transducer'])

    OUTFILE.write('\nProposed shadow cancellation circuit\n')
    for el in shadowguide:
        OUTFILE.write('>'
            + el
            + '\n'
            + shadow[el]
            +'\n')

seqs_preit = {}

#T7p sequence
seqs_preit['T7p'] = 'GCGCTAATACGACTCACTATAGG'

#Define initial input
try:
    USERINPUT = sys.argv[1]
except:
    USERINPUT = input(
        'Enter your input: '
        ).upper()

#Checks if input is a raw sequence or a fasta file
if USERINPUT.lower().split('.')[-1] == 'fasta':
    insequences = fileinput()

    for el in insequences:
        GUIDE[0] = el
        seqs_preit[el] = insequences[el]
        main()

        for name in GUIDE[1:-1]:
            del seqs_preit[name]
else:
    cmdinput()
    main()

#NuPACK files cleanup
#subprocess.call(
#    'rm -r /home/lugoibel/nupack3.2.2/python/tmp*',
#    shell=True)

elapsed_time = str(
    (time.time() - start_time)/60)

print('Job finished on '
    + str(datetime.datetime.now()).split('.')[0]
    + '. Elapsed time was: '
```

Annex I: *Python code of the developed algorithm (continues on the next page)*

```
    + elapsed_time[:-13]
    + ' minutes.')
```

***Annex I**: Python code of the developed algorithm*

Annex II: Code of the Nupack wrapper employed in this work, courtesy of Salis *et al.* (2009). Note that some modifications to the original wrapper have been performed with the aim of a proper performance along with the algorithm.

```
#Python wrapper for NUPACK 2.0 by Dirks, Bois, Schaeffer, Winfree, and Pierce (S
IAM Review)

#This file is part of the Ribosome Binding Site Calculator.

#The Ribosome Binding Site Calculator 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.

#The Ribosome Binding Site Calculator 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.

#You should have received a copy of the GNU General Public License
#along with Ribosome Binding Site Calculator.  If not, see <http://www.gnu.org/l
icenses/>.

#This Python wrapper is written by Howard Salis. Copyright 2008-
2009 is owned by the University of California Regents. All rights reserved. :)
#Use at your own risk.

import os.path
import os, subprocess, time, random, string

tempdir = "/tmp" + "".join([random.choice(string.digits) for x in range(6)])

current_dir = os.path.dirname(os.path.realpath(__file__)) + tempdir
if not os.path.exists(current_dir): os.mkdir(current_dir)

nupackbin_dir = "/home/lugoibel/nupack3.2.2/bin/"

debug = 0

#Class that encapsulates all of the functions from NuPACK 2.0


class NuPACK(dict):
    debug_mode = 0
    RT = 0.61597  # Gas constant times 310 Kelvin (in units of kcal/mol).

    def __init__(self, Sequence_List, material):

        self.ran = 0

        import re
        import string

        exp = re.compile('[ATGCU?&]', re.IGNORECASE)

        for seq in Sequence_List:
            if exp.match(seq)  == None:
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
                error_string = "Invalid letters found in inputted sequences." \

                               " Only ATGCU allowed. \n Sequence is \"" + \
                               str(seq) + "\"."
                raise ValueError(error_string)

        if not material == 'rna' and not material == 'dna' \
                and not material == "rna1999":
            raise ValueError("The energy model must be specified as "
                             "either ""dna"", ""rna"", or ""rna1999"" .")

        self["sequences"] = Sequence_List
        self["material"] = material

        random.seed(time.time())
        long_id = "".join([random.choice(string.ascii_lowercase + string.digits)
 for x in range(10)])
        self.prefix = current_dir + "/nu_temp_" + long_id

    def complexes(self, MaxStrands, Temp=37.0, ordered="", pairs="", mfe="",
                  degenerate="", dangles="some", timeonly="", quiet="",
                  AdditionalComplexes=[]):
        """A wrapper for the complexes command, which calculates the
        equilibrium probability of the formation of a multi-strand RNA or DNA
        complex with a user-defined maximum number of strands.
        Additional complexes may also be included by the user."""

        if Temp <= 0: raise ValueError("The specified temperature must be "
                                       "greater than zero.")
        if int(MaxStrands) <= 0:
            raise ValueError("The maximum number of strands must be greater"
                             " than zero.")

        #Write input files
        self._write_input_complexes(MaxStrands, AdditionalComplexes)

        #Set arguments
        material = self["material"]
        if ordered: ordered = " -ordered "
        if pairs: pairs = " -pairs "
        if mfe: mfe = " -mfe "
        if degenerate: degenerate = " -degenerate "
        if timeonly: timeonly = " -timeonly "
        if quiet: quiet = " -quiet "
        dangles = "-dangles " + dangles + " "


        #Call NuPACK C programs
        cmd = nupackbin_dir + "complexes"
        args = " -T " + str(Temp) + " -material " + material + " " + ordered \
               + pairs + mfe + degenerate + dangles + timeonly + \
               quiet + " "

        file = self.prefix
        #file = file[-2:]
        #file = str(file[0]) + "/" + str(file[1])
        output = subprocess.call(cmd + args + file, shell=True)

        self._read_output_ocx()
        if mfe:
            self._read_output_ocx_mfe()
            self._cleanup("ocx-mfe")
        #self._cleanup("ocx")
        #self._cleanup("ocx-key")

        self._cleanup("in")
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        #print "Complex energies and secondary structures calculated."
        self.ran = 1
        self["program"] = "complexes"

    def concentrations(self, concentrations="", quiet="", sort="",
                       cutoffvalue=0.001):
        if quiet:
            quiet = " -quiet"
        if sort != "":
            sort = " -sort " + str(sort)
        cutoffvalue = " -cutoffvalue" + str(cutoffvalue) + " "

        self._write_input_concentrations(concentrations)

        cmd = nupackbin_dir + "concentrations"
        args = quiet + sort + cutoffvalue
        output = subprocess.call(cmd + args + self.prefix, shell=True)

        self._read_output_con()
        self._cleanup("ocx")
        self._cleanup("ocx-key")
        self._cleanup("eq")
        self._cleanup("con")

    def prob(self, multi="-multi "):
        self.mfe([1, 2])
        self._write_input_prob()

        cmd =nupackbin_dir + "prob "
        args = multi + "-material " + self["material"] + " "
        result = subprocess.run(cmd + args + self.prefix, shell=True,
                                stdout=subprocess.PIPE)
        inf = str(result.stdout)
        inf = inf.split("\\n")
        prob = float(inf[-2])
        return prob


    def mfe(self, strands, Temp=37.0, multi=" -multi", pseudo="",
            degenerate="", dangles="some"):

        self["mfe_composition"] = strands

        if Temp <= 0:
            raise ValueError("The specified temperature must be "
                             "greater than zero.")

        if multi == 1 and pseudo == 1:
            raise ValueError("The pseudoknot algorithm does not work with "
                             "the -multi option.")

        #Write input files
        self._write_input_mfe(strands)

        #Set arguments
        material = self["material"]
        if multi == "":
            multi = ""
        if pseudo:
            pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "

        #Call NuPACK C programs
        cmd = nupackbin_dir + "mfe"
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        args = " -T " + str(Temp) + multi + pseudo + " -material " + \
               material + degenerate + dangles + " "
        output = subprocess.call(cmd + args + self.prefix, shell=True)

        self._read_output_mfe()
        self._cleanup("mfe")
        self._cleanup("in")
        self["program"] = "mfe"

    def subopt(self, strands, energy_gap, Temp=37.0, multi=" -multi",
               pseudo="", degenerate="", dangles="some"):

        self["subopt_composition"] = strands

        if Temp <= 0: raise ValueError("The specified temperature "
                                       "must be greater than zero.")

        if multi == 1 and pseudo == 1:
            raise ValueError("The pseudoknot algorithm does not work "
                             "with the -multi option.")

        #Write input files
        self._write_input_subopt(strands, energy_gap)

        #Set arguments
        material = self["material"]
        if multi == "": multi = ""
        if pseudo: pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "

        #Call NuPACK C programs
        cmd = nupackbin_dir + "subopt"
        args = " -T " + str(Temp) + multi + pseudo + " -material " +\
               material + degenerate + dangles + " "
        output = subprocess.call(cmd + args + self.prefix, shell=True)

        self._read_output_subopt()
        self._cleanup("subopt")
        self._cleanup("in")
        self["program"] = "subopt"

        #print "Minimum free energy and suboptimal secondary structures have bee
n calculated."

    def energy(self, strands, base_pairing_x, base_pairing_y, Temp=37.0,
               multi=" -multi", pseudo="", degenerate="", dangles="some"):

        self["energy_composition"] = strands

        if Temp <= 0:raise ValueError("The specified temperature must be"
                                      " greater than zero.")

        if multi == 1 and pseudo == 1:
            raise ValueError("The pseudoknot algorithm does not work "
                             "with the -multi option.")

        #Write input files
        self._write_input_energy(strands, base_pairing_x, base_pairing_y)

        #Set arguments
        material = self["material"]
        if multi == "": multi = ""
        if pseudo: pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        #Call NuPACK C programs
        cmd = nupackbin_dir + "energy"  # Imprime el resultado por pantalla.
        args = " -T " + str(Temp) + multi + pseudo + " -material " + \
               material + degenerate + dangles + " "

        output = subprocess.call(cmd + args + self.prefix + ">" + self.prefix
                                 + ".en", shell=True, stdout=True)

        file = open(str(self.prefix) + ".en")
        lectura = file.readlines()
        for line in lectura:
            line = line.strip("\n")
            if line[0] != "%":
                energy = float(line)
        file.close()

        self["energy_energy"] = []
        self["program"] = "energy"
        self["energy_energy"].append(energy)
        self["energy_basepairing_x"] = [base_pairing_x]
        self["energy_basepairing_y"] = [base_pairing_y]
        self._cleanup("in")
        self._cleanup("en")

        return energy

    def pfunc(self, strands, Temp=37.0, multi=" -multi", pseudo="",
              degenerate="", dangles="some"):

        self["pfunc_composition"] = strands

        if Temp <= 0: raise ValueError("The specified temperature must be "
                                       "greater than zero.")

        if multi == 1 and pseudo == 1:
            raise ValueError("The pseudoknot algorithm does not work "
                             "with the -multi option.")

        #Write input files
        #Input for pfunc is the same as mfe
        self._write_input_mfe(strands)

        #Set arguments
        material = self["material"]
        if multi == "": multi = ""
        if pseudo: pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "

        #Call NuPACK C programs
        cmd = nupackbin_dir + "pfunc"
        args = " -T " + str(Temp) + multi + pseudo + " -material " + \
               material + degenerate + dangles + " "

        output = subprocess.call(cmd + args + self.prefix + ">" + self.prefix +

                                 ".func", shell=True, stdout=True)

        file = open(str(self.prefix) + ".func")
        lectura = file.readlines()
        inf = []
        for line in lectura:
            line = line.strip("\n")
            if line[0] != "%" and line[0] != "Attempting":
                inf.append(float(line))
```

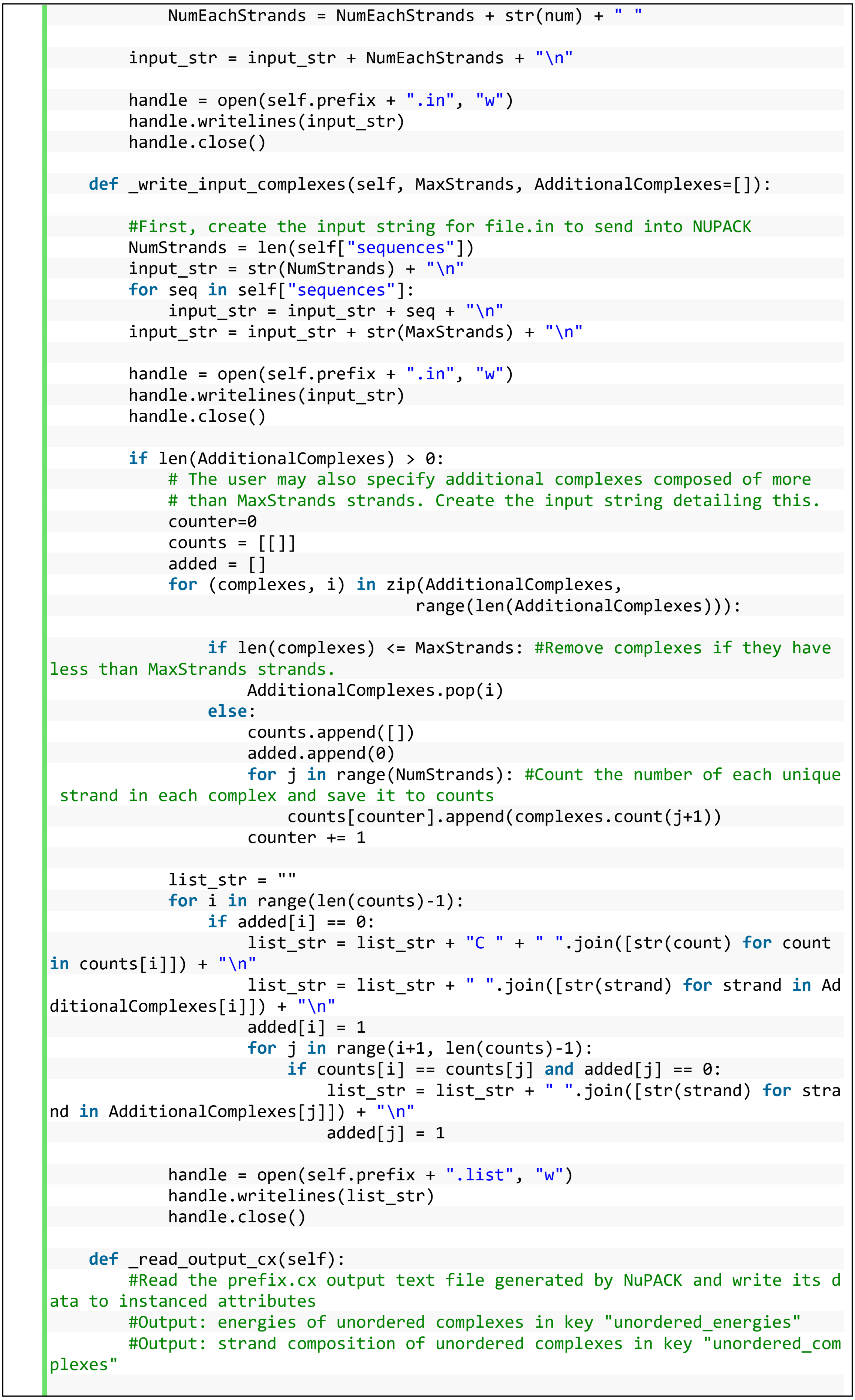
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        file.close()

        energy = inf[0]
        partition_function = float(inf[1])

        self["program"] = "pfunc"
        self["pfunc_energy"] = energy
        self["pfunc_partition_function"] = partition_function
        self._cleanup("in")
        self._cleanup("func")

        return partition_function

    def count(self, strands, Temp=37.0, multi=" -multi", pseudo="",
              degenerate="", dangles="some"):

        self["count_composition"] = strands

        if multi == 1 and pseudo == 1:
            raise ValueError("The pseudoknot algorithm does not work "
                             "with the -multi option.")

        #Write input files
        #Input for count is the same as mfe
        self._write_input_mfe(strands)

        #Set arguments
        material = self["material"]
        if multi == "": multi = ""
        if pseudo: pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "

        #Call NuPACK C programs
        cmd = nupackbin_dir + "count"
        args = " -T " + str(Temp) + multi + pseudo + " -material " + \
               material + degenerate + dangles + " "

        output = subprocess.call(cmd + args + self.prefix + ">" + self.prefix +

                                 ".count", shell=True)

        file = open(str(self.prefix) + ".count")
        lecture = file.readlines()
        for line in lecture:
            line = line.strip("\n")
            if line[0] != "%" and line[0] != "Attempting":
                number = float(line)


        self["program"] = "count"
        self["count_number"] = number
        self._cleanup("in")
        self._cleanup("count")

        return number

    def _write_input_prob(self):
        self._write_input_mfe([1, 2])
        handle = open(self.prefix + ".in", "a")
        handle.write(str(self["structure"]))
        handle.close()

    def _write_input_concentrations(self, concentrations):

        handle = open(self.prefix + ".con", "w")
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        number = len(self["sequences"])
        if concentrations == "":
            conc = "1e-6"
            handle.write((str(conc) + "\n") * number)
        else:
            for i in range(number):
                handle.write(str(concentrations[i]) + "\n")
        handle.close()

    def _write_input_energy(self, strands, base_pairing_x, base_pairing_y):
        """Creates the input file for energy NUPACK functions
        strands is a list containing the number of each strand in the complex
        (assumes -multi flag is used) base_pairing_x and base_pairing_y is a
        list of base pairings of the strands s.t. #x < #y are base paired. """

        NumStrands = len(self["sequences"])
        input_str = str(NumStrands) + "\n"
        for seq in self["sequences"]:
            input_str = input_str + seq + "\n"

        NumEachStrands = ""
        for num in strands:
            NumEachStrands = NumEachStrands + str(num) + " "

        input_str = input_str + NumEachStrands + "\n"
        for pos in range(len(base_pairing_x)):
            input_str = input_str + str(base_pairing_x[pos]) + "\t" + \
                        str(base_pairing_y[pos]) + "\n"

        handle = open(self.prefix + ".in", "w")
        handle.writelines(input_str)
        handle.close()

    def _write_input_subopt(self, strands, energy_gap):
        """Creates the input file for mfe and subopt NUPACK functions
        strands is a list containing the number of each strand in the complex
        (assumes -multi flag is used). """

        NumStrands = len(self["sequences"])
        input_str = str(NumStrands) + "\n"
        for seq in self["sequences"]:
            input_str = input_str + seq + "\n"

        NumEachStrands = ""
        for num in strands:
            NumEachStrands = NumEachStrands + str(num) + " "

        input_str = input_str + NumEachStrands + "\n"
        input_str = input_str + str(energy_gap) + "\n"

        handle = open(self.prefix + ".in", "w")
        handle.writelines(input_str)
        handle.close()

    def _write_input_mfe(self, strands):
        """ Creates the input file for mfe and subopt NUPACK functions
        strands is a list containing the number of each strand in the complex
        (assumes -multi flag is used). """

        NumStrands = len(self["sequences"])
        input_str = str(NumStrands) + "\n"
        for seq in self["sequences"]:
            input_str = input_str + seq + "\n"

        NumEachStrands = ""
        for num in strands:
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*



```
            NumEachStrands = NumEachStrands + str(num) + " "

        input_str = input_str + NumEachStrands + "\n"

        handle = open(self.prefix + ".in", "w")
        handle.writelines(input_str)
        handle.close()

    def _write_input_complexes(self, MaxStrands, AdditionalComplexes=[]):

        #First, create the input string for file.in to send into NUPACK
        NumStrands = len(self["sequences"])
        input_str = str(NumStrands) + "\n"
        for seq in self["sequences"]:
            input_str = input_str + seq + "\n"
        input_str = input_str + str(MaxStrands) + "\n"

        handle = open(self.prefix + ".in", "w")
        handle.writelines(input_str)
        handle.close()

        if len(AdditionalComplexes) > 0:
            # The user may also specify additional complexes composed of more
            # than MaxStrands strands. Create the input string detailing this.
            counter=0
            counts = [[]]
            added = []
            for (complexes, i) in zip(AdditionalComplexes,
                                         range(len(AdditionalComplexes))):

                if len(complexes) <= MaxStrands: #Remove complexes if they have less than MaxStrands strands.
                    AdditionalComplexes.pop(i)
                else:
                    counts.append([])
                    added.append(0)
                    for j in range(NumStrands): #Count the number of each unique strand in each complex and save it to counts
                        counts[counter].append(complexes.count(j+1))
                    counter += 1

            list_str = ""
            for i in range(len(counts)-1):
                if added[i] == 0:
                    list_str = list_str + "C " + " ".join([str(count) for count in counts[i]]) + "\n"
                    list_str = list_str + " ".join([str(strand) for strand in AdditionalComplexes[i]]) + "\n"
                    added[i] = 1
                    for j in range(i+1, len(counts)-1):
                        if counts[i] == counts[j] and added[j] == 0:
                            list_str = list_str + " ".join([str(strand) for strand in AdditionalComplexes[j]]) + "\n"
                            added[j] = 1

            handle = open(self.prefix + ".list", "w")
            handle.writelines(list_str)
            handle.close()

    def _read_output_cx(self):
        #Read the prefix.cx output text file generated by NuPACK and write its data to instanced attributes
        #Output: energies of unordered complexes in key "unordered_energies"
        #Output: strand composition of unordered complexes in key "unordered_complexes"
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        handle = open(self.prefix+".cx", "rU")

        line = handle.readline()

        #Read some useful data from the comments of the text file
        while line[0] == "%":

            words=line.split()

            if len(words) > 7 and words[1] == "Number" and words[2] == "of" \
                    and words[3] == "complexes" and words[4] == "from" \
                    and words[5] == "enumeration:":
                self["numcomplexes"] = int(words[6])

            elif len(words) > 8 and words[1] == "Total" \
                    and words[2] == "number" and words[3] == "of" \
                    and words[4] =="permutations" and words[5] == "to" \
                    and words[6] == "calculate:":
                self["num_permutations"] = int(words[7])

            line = handle.readline()

        self["unordered_energies"] = []
        self["unordered_complexes"] = []
        self["unordered_composition"] = []

        while line:
            words = line.split()

            if not words[0] == "%":

                complex = words[0]
                strand_compos = [int(f) for f in words[1:len(words)-1]]
                energy = float(words[len(words)-1])

                self["unordered_complexes"].append(complex)
                self["unordered_energies"].append(energy)
                self["unordered_composition"].append(strand_compos)

            line = handle.readline()
        handle.close()

    def _read_output_ocx(self):

    #Read the prefix.ocx output text file generated by NuPACK and write its data
 to instanced attributes
    #Output: energies of ordered complexes in key "ordered_energies"
    #Output: number of permutations and strand composition of ordered complexes
in key "ordered_complexes"

        handle = open(self.prefix+".ocx", "rU")

        line = handle.readline()

        #Read some useful data from the comments of the text file
        while line[0] == "%":

            words = line.split()

            if len(words) > 7 and words[1] == "Number" and words[2] == "of" \
                    and words[3] == "complexes" and words[4] == "from" \
                    and words[5] == "enumeration:":
                self["numcomplexes"] = int(words[6])

            elif len(words) > 8 and words[1] == "Total" \
                    and words[2] == "number" and words[3] == "of" \
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
                    and words[4] =="permutations" and words[5] == "to" \
                    and words[6] == "calculate:":
                self["num_permutations"] = int(words[7])

            line = handle.readline()

        self["ordered_complexes"] = []
        self["ordered_energies"] = []
        self["ordered_permutations"] = []
        self["ordered_composition"] = []

        while line:
            words = line.split()
            if not words[0] == "%":
                complex = words[0]
                permutations = words[1]
                strand_compos = [int(f) for f in words[2:len(words)-1]]
                energy = float(words[len(words)-1])

                self["ordered_complexes"].append(complex)
                self["ordered_permutations"].append(permutations)
                self["ordered_energies"].append(energy)
                self["ordered_composition"].append(strand_compos)

            line = handle.readline()
        handle.close()

    def _read_output_ocx_mfe(self):
    #Read the prefix.ocx output text file generated by NuPACK and write its data
 to instanced attributes
    #Output: energy of mfe of each complex in key "ordered_energy"


        #Make sure that the ocx file has already been read.
        if not (self.has_key("ordered_complexes")
                and self.has_key("ordered_permutations")
                and self.has_key("ordered_energies")
                and self.has_key("ordered_composition")):
            self._read_output_ocx(self.prefix)

        handle = open(self.prefix+".ocx-mfe", "rU")

        #Skip the comments of the text file.

        line = handle.readline()
        while line[0] == "%":
            line = handle.readline()

        self["ordered_basepairing_x"] = []
        self["ordered_basepairing_y"] = []
        self["ordered_energy"] = []
        self["ordered_totalnt"]=[]

        while line:
            words = line.split()

            if not line == "\n" and not words[0] == "%" and not words[0] == "":


                #Read the line containing the number of total nucleotides in the
 complex
                totalnt = words[0]

                self["ordered_totalnt"].append(totalnt)

                #Read the line containing the mfe
```
Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
                words = handle.readline().split()
                mfe = float(words[0])

                self["ordered_energy"].append(mfe)

                #Skip the line containing the dot/parens description of the secondary structure
                line = handle.readline()

                #Read in the lines containing the base pairing description of the secondary structure
                #Continue reading until a % comment
                bp_x = []
                bp_y = []

                line = handle.readline()
                words = line.split()
                while not line == "\n" and not words[0] == "%":
                    bp_x.append(int(words[0]))
                    bp_y.append(int(words[1]))
                    words = handle.readline().split()

                self["ordered_basepairing_x"].append(bp_x)
                self["ordered_basepairing_y"].append(bp_y)

            line = handle.readline()
        handle.close()

    def _read_output_con(self):
        handle = open(self.prefix + ".eq", "rU")
        inf = []
        for line in handle.readlines():
            if line[0] != "%":
                line = line.strip("\n")
                line = line.split("\t")
                line = line[2:-1]
                inf.append(line)
        self["complexes_concentrations"] = inf
        handle.close()



    def _read_output_mfe(self):
    #Read the prefix.mfe output text file generated by NuPACK and write its data to instanced attributes
    #Output: total sequence length and minimum free energy
    #Output: list of base pairings describing the secondary structure

        handle = open(self.prefix + ".mfe", "rU")

        #Skip the comments of the text file
        file = handle.readlines()
        text = []
        for line in file:
            if line[0] != "%" and line[0] != "" and line[0] != "\n":
                line = line.strip("\n")
                text.append(line)

        handle.close()
        self["mfe_basepairing_x"] = []
        self["mfe_basepairing_y"] = []
        self["mfe_energy"] = float(text[1])
        self["totalnt"] = int(text[0])
        self["structure"] = text[2]

        bp_x = []
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        bp_y = []

        for line in text[3:]:
            line = line.split("\t")
            bp_x.append(int(line[0]))
            bp_y.append(int(line[1]))

        self["mfe_basepairing_x"].append(bp_x)
        self["mfe_basepairing_y"].append(bp_y)


    def _read_output_subopt(self):
    #Read the prefix.subopt output text file generated by NuPACK and write its data to instanced attributes
    #Output: total sequence length and minimum free energy
    #Output: list of base pairings describing the secondary structure

        handle = open(self.prefix+".subopt", "rU")

        #Skip the comments of the text file
        line = handle.readline()
        while line[0] == "%":
            line = handle.readline()

        self["subopt_basepairing_x"] = []
        self["subopt_basepairing_y"] = []
        self["subopt_energy"] = []
        self["totalnt"]=[]

        counter = 0

        while line:
            words = line.split()

            if not line == "\n" and not words[0] == "%" and not words[0] == "":


                #Read the line containing the number of total nucleotides in the complex
                totalnt = words[0]

                self["totalnt"].append(totalnt)
                counter += 1

                #Read the line containing the mfe
                words = handle.readline().split()
                mfe = float(words[0])

                self["subopt_energy"].append(mfe)

                #Skip the line containing the dot/parens description of the secondary structure
                line = handle.readline()

                #Read in the lines containing the base pairing description of the secondary structure
                #Continue reading until a % comment
                bp_x = []
                bp_y = []

                line = handle.readline()
                words = line.split()
                while not line == "\n" and not words[0] == "%":
                    bp_x.append(int(words[0]))
                    bp_y.append(int(words[1]))
                    words = handle.readline().split()
```

Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
                self["subopt_basepairing_x"].append(bp_x)
                self["subopt_basepairing_y"].append(bp_y)

            line = handle.readline()
        handle.close()

        self["subopt_NumStructs"] = counter

    def _cleanup(self, suffix):

        if os.path.exists(self.prefix+"."+suffix):
            os.remove(self.prefix+"."+suffix)

        return

    def export_PDF(self, complex_ID, name="", filename="temp.pdf",
                   program=None):
        """Uses Zuker's sir_graph_ng and ps2pdf.exe to convert a secondary
        structure described in .ct format to a PDF of the RNA."""

        if program is None:
            program = self["program"]

        inputfile = "temp.ct"
        self.Convert_to_ct(complex_ID, name, inputfile, program)


        cmd = "sir_graph_ng" #Assumes it's on the path
        args = "-p" #to PostScript file
        output = popen2.Popen3(cmd + " " + args + " " + inputfile, "r")
        output.wait()
        if debug == 1:
            print(output.fromchild.read())

        inputfile = inputfile[0:len(inputfile)-2] + "ps"

        cmd = "ps2pdf" #Assumes it's on the path
        output = popen2.Popen3(cmd + " " + inputfile, "r")
        output.wait()
        if debug == 1:
            print(output.fromchild.read())

        outputfile = inputfile[0:len(inputfile)-2] + "pdf"

        #Remove the temporary file "temp.ct" if it exists
        if os.path.exists("temp.ct"): os.remove("temp.ct")

        #Remove the temporary Postscript file if it exists
        if os.path.exists(inputfile): os.remove(inputfile)

        #Rename the output file to the desired filename.
        if os.path.exists(outputfile): os.rename(outputfile,filename)
        #Done!

    def Convert_to_ct(self, complex_ID, name, filename="temp.ct",
                      program="ordered"):
        """Converts the secondary structure of a single complex into the
        .ct file format, which is used with sir_graph_ng (or other programs)
        to create an image of the secondary structure."""

        #hacksy way of reading from data produced by 'complex', by 'mfe', or by
'subopt'
        data_x = program + "_basepairing_x"
        data_y = program + "_basepairing_y"
        mfe_name = program + "_energy"
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        composition_name = program + "_composition"

        #Format of .ct file

        #Header: <Total # nt> \t dG = <# mfe> kcal/mol \t <name of sequence>
        #The Rest:
        #<nt num> \t <bp letter> \t <3' neighbor> \t <5' neighbor> \t <# of bp'ing, 0 if none> \t ...
        #<strand-specific nt num> \t <3' neighbor if connected by helix> \t <5' neighbor if connected by helix>

        #Extract the data for the desired complex using complex_ID
        bp_x = self[data_x][complex_ID]
        bp_y = self[data_y][complex_ID]
        mfe = self[mfe_name][complex_ID]

        if program == "mfe" or program == "subopt" or program == "energy":
            composition = self[composition_name]
        elif program == "ordered" or program == "unordered":
            composition = self[composition_name][complex_ID]


        #Determine concatenated sequence of all strands, their beginnings, and ends
        allseq = ""
        strand_begins = []
        strand_ends = []

        #Seemingly, the format of the composition is different for the program complex vs. mfe/subopt
        #for mfe/subopt, the composition is the list of strand ids
        #for complex, it is the number of each strand (in strand id order) in the complex
        #for mfe/subopt, '1 2 2 3' refers to 1 strand of 1, 2 strands of 2, and 1 strand of 3.
        #for complex, '1 2 2 3' refers to 1 strand of 1, 2 strands of 2, 2 strands of 3, and 3 strands of 4'.
        #what a mess.

        if program == "mfe" or program == "subopt" or program == "energy":
            for strand_id in composition:
                strand_begins.append(len(allseq) + 1)
                allseq = allseq + self["sequences"][strand_id-1]
                strand_ends.append(len(allseq))

        else:
            for (num_strands, strand_id) in \
                    zip(composition, range(len(composition))):
                for j in range(num_strands):
                    strand_begins.append(len(allseq) + 1)
                    allseq = allseq + self["sequences"][strand_id]
                    strand_ends.append(len(allseq))

        seq_len = len(allseq)

        #print "Seq Len = ", seq_len, "  Composition = ", composition
        #print "Sequence = ", allseq
        #print "Base pairing (x) = ", bp_x
        #print "Base pairing (y) = ", bp_y


        #Create the header
        header = str(seq_len) + "\t" + "dG = " + str(mfe) + " kcal/mol" \
                 + "\t" + name + "\n"
```

***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        #Open the file
        handle = open(filename,"w")

        #Write the header
        handle.write(header)

        #Write a line for each nt in the secondary structure
        for i in range(1, seq_len+1):
            for (nt, pos) in zip(strand_begins, range(len(strand_begins))):
                if i >= nt:
                    strand_id = pos


            #Determine 3' and 5' neighbor
            #If this is the beginning of a strand, then the 3' neighbor is 0
            #If this is the end of a strand, then the 5' neighbor is 0

            if i in strand_begins:
                nb_5p = 0
            else:
                nb_5p = i - 1

            if i in strand_ends:
                nb_3p = 0
            else:
                nb_3p = i + 1

            if i in bp_x or i in bp_y:
                if i in bp_x: nt_bp = bp_y[bp_x.index(i)]
                if i in bp_y: nt_bp = bp_x[bp_y.index(i)]
            else:
                nt_bp = 0

            #Determine strand-specific counter
            strand_counter = i - strand_begins[strand_id] + 1

            #Determine the 3' and 5' neighbor helical connectivity
            #If the ith nt is connected to its 3', 5' neighbor by a helix, then
include it
            #Otherwise, 0
            #Helix connectivity conditions:
            #The 5' or 3' neighbor is connected via a helix iff:
            #a) helix start: i not bp'd, i+1 bp'd, bp_id(i+1) - 1 is bp'd, bp_id
(i+1) + 1 is not bp'd
            #b) helix end: i not bp'd, i-1 bp'd, bp_id(i-
1) - 1 is not bp'd, bp_id(i-1) + 1 is bp'd
            #c) helix continued: i and bp_id(i)+1 is bp'd, 5' helix connection i
s bp_id(bp_id(i)+1)
            #d) helix continued: i and bp_id(i)-
1 is bp'd, 3' helix connection is bp_id(bp_id(i)-1)
            #Otherwise, zero.

            #Init
            hc_5p = 0
            hc_3p = 0

            if i in bp_x or i in bp_y:  # Helix continued condition (c,d).
                if i in bp_x: bp_i = bp_y[bp_x.index(i)]
                if i in bp_y: bp_i = bp_x[bp_y.index(i)]

                if bp_i+1 in bp_x or bp_i+1 in bp_y:  # Helix condition c.
                    if bp_i+1 in bp_x: hc_3p = bp_y[bp_x.index(bp_i+1)]
                    if bp_i+1 in bp_y: hc_3p = bp_x[bp_y.index(bp_i+1)]

                if bp_i-1 in bp_x or bp_i-1 in bp_y:  # Helix condition d.
                    if bp_i-1 in bp_x: hc_5p = bp_y[bp_x.index(bp_i-1)]
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
                    if bp_i-1 in bp_y: hc_5p = bp_x[bp_y.index(bp_i-1)]

            else: #helix start or end (a,b)

                if i+1 in bp_x or i+1 in bp_y:  # Start, condition a.
                    if i+1 in bp_x: bp_3p = bp_y[bp_x.index(i+1)]
                    if i+1 in bp_y: bp_3p = bp_x[bp_y.index(i+1)]

                    if bp_3p + 1 not in bp_x and bp_3p + 1 not in bp_y:
                        hc_3p = i + 1

                if i-1 in bp_x or i-1 in bp_y: #End, condition b
                    if i-1 in bp_x: bp_5p = bp_y[bp_x.index(i-1)]

                    if i-1 in bp_y: bp_5p = bp_x[bp_y.index(i-1)]

                    if bp_5p - 1 not in bp_x and bp_5p - 1 not in bp_y:
                        hc_5p = i - 1


            line = str(i) + "\t" + allseq[i-1] + "\t" + str(nb_5p) + "\t" + \
                   str(nb_3p) + "\t" + str(nt_bp) + "\t" + str(strand_counter) \

                   + "\t" + str(hc_5p) + "\t" + str(hc_3p) + "\n"

            handle.write(line)

        #Close the file. Done.
        handle.close()


if __name__ == "__main__":

    import re

    #sequences = ["AAGATTAACTTAAAAGGAAGGCCCCCCATGCGATCAGCATCAGCACTACGACTACGCGA","acctcctta","ACGTTGGCCTTCC"]
    sequences = ["AAGATTAACTTAAAAGGAAGGCCCCCCATGCGATCAGCATCAGCACTACGACTACGCGA"]


    #Complexes
    #Input: Max number of strands in a complex. Considers all possible combinations of strands, up to max #.
    #'mfe': calculate mfe? 'ordered': consider ordered or unordered complexes?
    #Other options available (see function)

    AddComplexes = []
    test = NuPACK(sequences,"rna1999")
    test.complexes(3, mfe=1, ordered=1)

    print(test)

    strand_compositions = test["ordered_composition"]
    num_complexes = len(strand_compositions)
    num_strands = len(sequences)

    for counter in range(num_complexes):
        output = "Complex #" + str(counter+1) + " composition: ("
        for strand_id in strand_compositions[counter][0:num_strands-1]:
            output = output + str(strand_id) + ", "
        output += str(strand_compositions[counter][num_strands-1]) + ")"

        output = output + "  dG (RT ln Q): " + \
                 str(test["ordered_energy"][counter]) + " kcal/mol"
        output = output + "  # Permutations: " + \
                 str(test["ordered_permutations"][counter])
```

Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        print(output)
        test.export_PDF(counter, name="Complex #" + str(counter+1),
                        filename="Complex_" + str(counter) + ".pdf",
                        program="ordered")

    #Mfe
    #Input: Number of each strand in complex.
    #Options include RNA/DNA model, temperature, dangles, etc. (See function).
    #Example: If there are 3 unique strands (1, 2, 3), then [1, 2, 3] is one of
each strand and [1, 1, 2, 2, 3, 3] is two of each strand.

    #test.mfe([1, 2], dangles = "all")
    #num_complexes = test["mfe_NumStructs"]  #Number of degenerate complexes (sa
me energy)
    #dG_mfe = test["mfe_energy"]
    #print "There are ", num_complexes, " configuration(s) with a minimum free e
nergy of ", dG_mfe, " kcal/mol.
```

Annex II: *Code of the Nupack wrapper (Salis et al., 2009)*