Quantum algorithm for RNA design

-

rapport de stage de L3 de biologie

Émile Larroque,
Département d'Enseignement et Recherche en Biologie / École Normale Supérieure de Paris-Saclay

Supervisor : Aritra Sarkar,
Department ot Quantum & Computer Enginneering / Technical University of Delft

June 12 to August 11 2023

Table des matières

1 Introduction

RNA RNA is a linear polymeric molecule whose monomers are nucleotides. The sequence of nucleotide residues (called nucleobases, or simply *bases*) constitutes the primary structure of an RNA molecule. However, RNA molecules are flexible and can fold, presenting more levels of structure.

Secondary structure In an RNA molecule, bases can interact with each other via weak bonds such as hydrogen bonds, forming structures that reduce the free energy of the RNA molecule and stabilise the folding. These structures are mainly base pairs (tow bases that are distant from each other in the sequence interacting via hydrogen bonds in a stabilising way) interacting with their neighbours in the sequence (stacking, dandling end, mismatch). However, there also exist such structures based on the interaction of three distant bases [Bro20] or four distant bases even though the biological significance of the latter is still subject to debates [TRG21]. These structures constitute the secondary structure of an RNA molecule.

In this work, we only consider secondary structures involving base pairs. Yet simplifying, we argue that :

- this hypothesis is less simplifying than what has been considered until now in quantum computing approaches to the folding problem of RNA and the inverse folding problem of RNA (see Section 4.1),
- the problem considered here is still of biological interest :
  - most RNA secondary structures for which biologists would design an RNA sequence, would only consist in base pairs helices and unpaired regions,
  - even though ignoring certain secondary structures could lead to false positives (designing a sequence that would fold into one of these ignored structures instead of the desired one), base pairs helices and unpaired regions are still the most common types of RNA secondary structures and false positives due to this simplification would arguably be rare.

Inverse folding problem While the folding problem of RNA (Problem 1) consists in predicting how an RNA molecule of a given sequence will fold, the inverse folding problem of RNA (Problem 2) consists in finding a sequence that would make the RNA molecule fold with the desired secondary structure [Hof+98].

Problem 1 (folding problem of RNA). *For all $n \in \mathbb{N}$, let S_n be the set of RNA sequences of length n and F_n be the set of secondary structures of RNA molecules of length n. The folding problem of RNA is :*
Input : *$n \in \mathbb{N}$ and $s_0 \in S_n$.*
Output : *$f_0 \in F_n$ such that an RNA molecule of sequence s_0 folds into the secondary structure f_0.*

Problem 2 (inverse folding problem of RNA). *For all $n \in \mathbb{N}$, let S_n be the set of RNA sequences of length n and F_n be the set of secondary structures of RNA molecules of length n. The inverse folding problem of RNA is :*
Input : *$n \in \mathbb{N}$ and $f_0 \in F_n$.*
Output : *$s_0 \in S_n$ such that an RNA molecule of sequence s_0 folds into the secondary structure f_0.*

In this work, we are interested in the inverse folding problem.

Many biologists would like to be able to design RNA that have a particular desired biological function (i.e. enzymatic function, structural function, etc.). It would be a fundamental molecular biology tool, that would allow deeper study of cellular pathways, or even control over it [MWS16].

As stated in Problem 2, the problem is not entirely well defined yet, since it still lacks a folding model. We will consider the standard free energy model called nearest neighbour model [TM10] (see Section 2.1.2). This will give us a free energy function E such that $E(s, f)$ is the free energy of an RNA molecule of sequence s and secondary structure f. Then, the folding problem consists in energy minimisation (Problem 3).

Problem 3 (folding problem of RNA with free energy minimisation). *For all $n \in \mathbb{N}$, let S_n be the set of RNA sequences of length n and F_n be the set of secondary structures of RNA molecules of length n. Let $E : \bigcup_{n\in\mathbb{N}} S_n \times F_n \to \mathbb{R}$ be the free energy function of RNA. The folding problem of RNA is :*
Input : *$n \in \mathbb{N}$ and $s_0 \in S_n$.*
Output : *$f_0 \in F_n$ such that $f_0 \in \underset{f\in F_n}{argmin}(E(s_0, f))$.*

And then follows the definition of the inverse folding problem (Problem 4).

Problem 4 (inverse folding problem of RNA with free energy minimisation). *For all $n \in \mathbb{N}$, let S_n be the set of RNA sequences of length n and F_n be the set of secondary structures of RNA molecules of length n. Let $E : \bigcup_{n\in\mathbb{N}} S_n \times F_n \to \mathbb{R}$ be the free energy function of RNA. The inverse folding problem of RNA is :*
Input : *$n \in \mathbb{N}$ and $f_0 \in F_n$.*
Output : *$s_0 \in S_n$ such that $f_0 \in \underset{f\in F_n}{argmin}(E(s_0, f))$.*

Simplifying the problem Yet formal, this exercise of precise problem definition avoids considering the wrong problem, such as Problem 5.

Problem 5 (over-simplified inverse folding problem). *For all $n \in \mathbb{N}$, let S_n be the set sequences of length n and F_n be the set of secondary structures of length n. Let $E : \bigcup_{n\in\mathbb{N}} S_n \times F_n \to \mathbb{R}$ be the free energy function of RNA. The simplified inverse folding problem is :*
Input : *$n \in \mathbb{N}$ and $f_0 \in F_n$.*
Output : *$s_0 \in \underset{s\in S_n}{argmin}(E(s, f_0))$.*

Indeed, Problem 5 is a simplification of the inverse folding problem that can be found in the literature of quantum approaches to inverse folding problems. Indeed, it can be useful when dealing with proteins, where a residue can interact with many other residues at the same time [Kha+23]. But with RNA, it would just lead to compute a sequence s_0 such that every base pair in f_0 is a GC pair (since GC pairs reduce more the free energy than AU pairs). Not only making the problem computationally trivial (vanishing the need for quantum computing), the simplified Problem 5 over RNA would almost systematically lead to major false positives (sequences leading to RNA molecules that do not fold in test tubes into the desired secondary structure at all).

So in this work, we consider the exact Problem 4 and simplifications only occur in the parameters of the problem :

— the set of candidate sequences : sequences only contain A, C, G and U bases with no modification,
— the set of candidate secondary structures : only anti-parallel Watson-Crick base pairs, no pseudoknot,
— the free energy function : see Section 2.1.2.

These simplifications aim :

— to fundamentally reduce the space complexity of the algorithm ($\mathcal{O}(n)$ qbits instead of $\mathcal{O}(n \cdot log(n))$ and simplifying it : no pseudoknot
— to be able to use standard already made RNA free energy computation models such as the nearest neighbour model [TM10] : only the most standard A, C, G and U bases and only anti-parallel base pairs,
— to simplify the programming so that it is easier to get a first working version and to slightly reduce again the number of qbits so that it is possible so simulate parts of the program on small instances and thus testing it : only Watson-Crick base pairs (in particular no GU pairs) and other simplifications of the nearest neighbour model simplifying the energy computation (see Section 2.1.2).

As a result, the structure of the presented algorithm (see Section 2.2) could be used in a quite straightforward way to solve the problem with more realistic hypothesis. Only pseudoknots would need more involved algorithmic work to be included. In brief, the presented work aims to become a proof of concept that shows the relevance of quantum computing for solving the inverse folding problem of RNA in a biologically relevant way.

It is not yet, since the programming of the energy computation is not finished. (Every part of the algorithm the programming of which is not finished is precisely indicated in Section 2.2 where the algorithm is described.)

So, at that point, the formalisation of the problem has been pushed up to Problem 6. The fully formalised problem will be presented later in Section 2.

Problem 6 (inverse folding problem of RNA with bases). *For all $n \in \mathbb{N}$, let $S_n = \{ A, C, G, U \}^n$ be the set of RNA sequences of length n and F_n be the set of secondary structures of RNA molecules of length n. Let $E : \bigcup_{n\in\mathbb{N}} S_n \times F_n \to \mathbb{R}$ be the free energy function of RNA. The inverse folding problem of RNA is :*
Input : *$n \in \mathbb{N}$ and $f_0 \in F_n$.*
Output : *$s_0 \in S_n$ such that $f_0 \in \underset{f\in F_n}{argmin}(E(s_0, f))$.*

Quantum computing approach The inverse folding problem (Problem 6) is a complex combinatorial optimisation problem. (Note that the number of sequences is exponential in the length of the RNA molecule). Until now, there is no classical algorithm to solve it in time sublinear in the number of sequences. Only heuristics and machine learning-based methods are available to try to reduce the time of computation, with no guaranty in every case, and sometimes at the cost of not exactly solving the problem [Chu+18].

However in the future, quantum computing will have the potential to have an advantage when solving exactly that kind of problem, even though nowadays quantum computers are not advanced enough yet to solve big instances. The Grover algorithm provides a straightforward way to design a quantum algorithm solving the inverse folding problem of RNA in time $\mathcal{O}(\sqrt{N})$ where N is the number of sequences. Even though $\sqrt{N}$ is still exponential in the length of the RNA molecule, it represents a quadratic quantum advantage over classical algorithmic methods, so it may be key to solve bigger instances of the problem. That is the path we follow in the present work (see Section 2.1.4).

2 Material and methods

2.1 General framework

2.1.1 Dot-bracket notation

The dot-bracket notation is a standard way to represent RNA secondary structure (without the primary structure). It describes which base is paired with which other base, independently of the sequence :

- **(** represents a base that forms a base pair with a base downstream in the sequence,
- **)** represents a base that forms a base pair with a base upstream in the sequence,
- **.** represents an unpaired base.

Pseudoknot Let $s = (s_i)_{i\in[1;n]}$ be an RNA sequence. If s_i should pair with s_k, s_j should pair with s_l and $i < j < k < l$, this structure is called a pseudoknot.

When pseudoknots are allowed, the dot-bracket notation is ambiguous so its alphabet sould be adapted and become infinite (or with $\mathcal{O}(log(n))$ characters when the length n of the sequence is known). When pseudoknots are not allowed (which is the case in this work), a word in $\{\mathbf{(}, \mathbf{)}, \mathbf{.}\}^*$ is a valid dot-bracket notation describing a secondary structure if and only if it is well parenthesised.

2.1.2 Nearest neighbour model

The nearest neighbour model [TM10] is a reference RNA free energy computation model. It is for example used by the ViennaRNA package [Lor+11]. Based on experimental data, it assigns free energy contributions (either positive or negative) to specific patterns, and the predicted free energy of a given association of a primary and a secondary structure of RNA is the sum of these contributions.

Loop Any base pair encloses a part of the sequence (everything between **(** and the matching **)**). And any base pair defines a loop associated to it, that is the enclosed unpaired region adjacent to this base pair and the potential base pairs closing this unpaired region.

Example 1 (the types of loop defined by examples)**.** *The following examplify the different types of loop :*

- **(.....)** *is a* hairpin loop *of size 5, it is composed of 7 bases. For steric reasons, hairpin loops cannot be of size 2 or less. Hairpin loop of size 4 or more with only C unpaired bases are a particular case that is ignored in this work (treated as any other hairpin loop).*
- **(..(...)....)** *is an* internal loop *of size 6, it is composed of 10 bases. . . . is any well parenthesised word, and is not part of the loop. The more different in size the tow unpaired region (here of size 2 and 4), the less stabilising this structure is.*
- **((...))** *is a* stack*. A stack always contains 4 bases. Physically, the tow base pairs interact and fit together. Multiple stacks form an helix.*
- **((...)..)** *and* **(..(...))** *are* bulge loop *of size 2, they are composed of 6 bases. In a bulge loop of size 1, the helix is not interrupted (the tow base pair fit with each other and the contribution of a stack applies).*
- **(..(...)...(...)..(...).....)** *is a* multibranch loop*, or simply* multiloop*, of size 12. It is composed of 20 bases. Around each base pair of the multiloop, the tow unpaired regions of the multiloop could be of different size. This difference in size is the asymmetry around the base pair. The more the average asymmetry, the slightly less stable is this structure (this contribution to free energy is ignored in the present work).*

Dandling ends When an unpaired base is adjacent to a base pair, it can stack on it in a stabilising way. This structure of three bases is called a *dandling end*. The dot-bracket configuration is **(....)** or **.(...)..**

Terminal mismatch When an unpaired base stacks on the one base of a base pair and another unpaired base stacks on the other base of the base pair (on the same side of the base pair), these four bases form what is called a *terminal mismatch*. This structure is more stabilising than a dandling end. The dot-bracket configuration is **(.....)** or **.(...).**.

Coaxial stacking When tow helices end are separated by no (respectively one) unpaired base, they can stack on each other directly (respectively with a mismatch mediating the stacking). This structure of four (respectively six bases) is called a *terminal mismatch* and is more stabilising than a terminal mismatch. Coaxial stacking are ignored in this work.

Contributions to free energy Stacks, dandling ends, terminal mismatches and coaxial stacking have energy contribution that depends on the bases involved. In addition to that, loops have energy contributions that depend on their type and size. Moreover, certain loops of limited size are particular cases with particular energy contribution (for example internal loops of size inferior to 4). The full list of parameters and particular cases used in this work can be found in the Nearest Neighbour model Data Base (NNBD, `https://rna.urmc.rochester.edu/NNDB/turner04/index.html`) [TM10].

2.1.3 Customised dot-bracket notation

As shown in the Section 2.1.2, the dot-bracket notation is insufficient to fully specify a secondary structure. Indeed, neither **(.....)** nor **.(...).** specify the dandling ends and terminal mismatches. (Nor does the dot-bracket notation specifies coaxial stacking.) Crucially, since each base can participate in at most one of these structures, there is a choice to do about it. And even though a coaxial stacking is more stabilising than a terminal mismatch, that itself is more stabilising than a dandling end, there is a non trivial choice to do : see Example **??**.

Example 2 (the choice of dandling ends and terminal mismatches is non trivial)**.** *In the bracket-dot configuration* **.(...).(...).(...).(...).(...).(...).(...).** *it is not clear which choice of dandling ends and terminal mismatches has the most stabilising free energy contribution. Actually, it depends on the sequence ! Moreover, and more importantly, this example shows that even when the dot-bracket configuration is specified, the choice of dandling ends and terminal mismatches that minimises the free energy cannot be known locally : it depends on the whole RNA molecule.*

By the way, if we allow coaxial stacking, this example is still non trivial : the best choice of coaxial stacking is non local.

As will be explained in Section 2.1.4 and section 2.2.2, we need a way to fully specify the secondary structure (without the primary structure). Thus, we will use a custom dot-bracket notation to specify the choice of dandling ends and terminal mismatches.

So **.** will now exist in tow flavours :
- **<** represents an unpaired base that is not stacked onto the base on its right,
- **>** represents an unpaired base that is not stacked onto the base on its left,
- **.** will still be used in this report to represent **>** or **<** without specifying which one.

Thus,
- **(<...<)**, **>(...)>**, **(>...>)** and **<(...)<** are dandling ends,
- **(<...>)** and **>(...)<** are terminal mismatches,
- **(>...<)** and **<(...)>** are end of helices with no dandling end nor terminal mismatch.

Thus, the final form of our formalisation exercise is Problem 7

Problem 7 (inverse folding problem fully specified). *For all $n \in \mathbb{N}$, let $S_n = \{ A, C, G, U \}^n$ be the set of RNA sequences of length n, let $F_n \subseteq \{ (,), <, > \}^n$ be the set of customised dot-bracket word of length n that describe a*

secondary structure. Let $E : \bigcup_{n\in\mathbb{N}} S_n \times F_n \to \mathbb{R}$ *be the free energy function of RNA. The inverse folding problem of RNA is :*
Input : $n \in \mathbb{N}$ *and* $f_0 \in F_n$.
Output : $s_0 \in S_n$ *such that* $f_0 \in \underset{f\in F_n}{argmin}(E(s_0, f))$.

Remark 1 (a priori wrong dandling end and terminal mismatch choice). *Since a terminal mismatch is more stabilising than a dandling end, that itself is more stabilising than a simple end of helix, we can tell that* **.<|** *and* **|>.** *are wrong patterns, or at least that these secondary structures would not be the most stable ones (where* **|** *represents* **(** *or* **)** *without specifying which one).*

Remark 2 (space efficiency of the customised dot-bracket notation). *As the classical dot-bracket notation, the customised one requires 2 bits (or qbits) per character.*

2.1.4 Grover search framework

Finding an element The Grover search [Gro96] is a quantum algorithm that enables to find an element s of a set S that satisfies a certain criterion in time $\mathcal{O}(\sqrt{|S|})$, as far as the *oracle* (the part of the quantum circuit that check if the criterion is fulfilled) is executed in time $\mathcal{O}(\sqrt{|S|})$.

Finding the minimum On this base, a variety of algorithms have been developed, in particular for finding the element that minimises a certain quantity computed by the oracle [DH99]. That is what will enable us to handle the $argmin()$ in Problem 7.

Initiation The initiation of an algorithm of the Grover search family consists in setting the quantum memory into a superposition (with positive real amplitude) of all the states representing candidate elements. Then, the oracle computes (in parallel thanks to the superposition) for each element an indicator (a criterion, a function of which we search for the minimum etc.) and the result is the superposition of the results. Then a well chosen operation called *diffuser* is applied $\mathcal{O}(\sqrt{|S|})$ times in order to set to a non zero amplitude the satisfying candidates and to a zero amplitude the other ones. Some algorithms of the Grover family require extra work to find the good diffuser (typically when the number of satisfying candidates is initially unknown).

In the present work, verifying that a customised dot-bracket word describes a secondary structure (bracket are well parenthesised, no hairpin loop of size 2 or less, no wrong pattern of dandling end and terminal mismatch choice) is a complicated task in itself. So the strategy is to initialise the part of the quantum memory dedicated to the secondary structure with a superposition of all customised dot-bracket words. (That is why the customised dot-bracket introduces in Section 2.1.3 is needed.) Then, the oracle has tow tasks :

- check if the customised dot-bracket word f describes a secondary structure,
- if it does, computing the energy of the association of the primary structure s and the secondary structure f (if it does not, computing any number since it will be ignored).

2.2 Algorithm

2.2.1 General structure of the search

Input The user gives **`specification_folding`** (a classical variable, registers are quantum registers in this text) that is a bracket-dot word describing a secondary structure (not necessarily a customised bracket-dot word, the secondary structure the user asks for can be under specified).

Definition 1 (configuration). *A configuration is a sequence* $s \in \{ A, C, G, U \}^n$ *and a customised bracket-dot word* $f \in \{ \mathbf{(}, \mathbf{)}, \mathbf{<}, \mathbf{>} \}^n$ *where* $n \in \mathbb{N}$ *is the length of the sequence.*

Definition 2 (valid sequence). *A sequence* $s \in \{ A, C, G, U \}^n$ *is valid if it produces only valid base pairs with respect to the* **`specification_folding`**.

Definition 3 (wrong configuration). *A configuration (s, f) is* wrong *if f does not describe a secondary structure, or if s is invalid, or if s produces invalid base pairs with respect to f. (Configurations that are not wrong are* correct.*) A* **`wrong`** *register is initialised to 0 and is incremented every time the configuration is detected to be wrong (incrementation is reversible).*

Initialisation Registers of the configurations are initialised this way :

- the **`sequence`** register is initialised to the superposition of all valid sequences s (thanks to **`specification_folding`**),
- the **`folding`** register is initialised to all the customised dot-bracket words f,
- the **`choice`** register (not programmed yet) is initialised to the superposition of all the missing choices of **`<`** and **`>`** where **`.`** in the **`specification_folding`** lead to ambiguity about terminal mismatches, dandling ends and simple helices ends. Together, **`choice`** and **`specification_folding`** encode a (superposition of) target fully specified secondary structure(s) f_0.

Oracle For each (s, f, f_0) in superposition, the oracle computes :

- the **`wrong`** register so that it is different from 0 if and only if (s, f) is a wrong configuration (see Sections 2.2.2 and 2.2.3),
- the **`energy`** register so that it is equal to $E(s, f)$ if the configuration (s, f) is right (and is equal to anything if the configuration (s, f) is wrong). (The programming of this is quite advances but not finished yet.)
- then it computes $E(s, f_0)$ and subtracts it to the **`energy`** register (actually subtract it piece by piece directly instead of using qbits to compute it before subtraction). at the end **`energy`** $= E(s, f) - E(s, f_0)$ (The programming of this is not finished at all.)

Variant of Grover search Apply a variant of Grover search in order to select s and f_0 (that is **`sequence`** and **`choice`**) such that $\forall f \in \{$**`folding`** $\mid$ **`wrong`** $= 0\}$, **`energy`** ≥ 0 (that is an RNA molecule with primary structure s would not preferentially fold into any other secondary structure f instead of the target secondary structure f_0).

Remark 3 (limit of my understanding). *I admit that I do not fully understand the Grover search family toolkit yet, but apparently making it work the way described above is not the difficult part at all. That is why the work presented here (and that I was asked to do) is the oracle.*

2.2.2 General structure of the oracle

Remark 4 (about code). *The code in Python produced during this internship can be found in Appendix A so that this report is as autonomous as possible. However copy-pasting code in order to execute it is not handy at all (moreover many long code lines are broken into several in Appendix A and would require direct editing if copy-pasted for execution).*

That is why the original interactive computing file in which the code has been organised is joined to this report : see Appendix B.

The quantum computing library used is Qiskit (`https://qiskit.org/`).
Finally, some important extracts of the code are included and commented in the body of the report.
Please also note that **`<`** *and* **`>`** *are written* **`<-.`** *and* **`.->`** *most of the time in the code.*

At each position **`k_left`** of the sequence :

- the **`folding`** is analysed. That is the role of the **`characterise_loop()`** function. If a loop starts at this position (that is if **`folding[k_left]`** encodes **`(`**) :
  - it checks if it has a matching **`)`** downstream. The result of this complicated test is stored in **`continue_searching_right_matching_left`**.

- it detects, computes and stores information into some registers that are used by the energy computation part
- if **folding[k_left]** encodes **(**, some energy contributions associated to this base pair and to the loop that starts there are added to the global **energy** register. That is the role of the **compute_energy()** function.
- after that, registers are cleaned by applying the inverse operations that set them, in opposite order. That is the role of the **inversed(characterise_loop)()** function.

In order to check if the **folding** is well parenthesised, it is not sufficient to check that **(** have matching **)** downstream. One should also check that **)** have matching **(** upstream. In order to do that, we imagine an extra **(** at the beginning of the **folding** and check that it *does not have* any **)** downstream. That is the role of **characterise_loop)(compute_energy=false)**.

See Listing 1.

```
def RNA_design(circuit,
               folding,sequence,count,continue_searching_right_matching_left,nb_enclosed,
    loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,
    right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,
    use_right_matching_left_next,incorrect_configuration,energy,
               all_registers,
               nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
               all_nb_bits,
               incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,
    incr_nb_enclosed,incr_energy,decr_energy,add_energy,
               all_gates,
               length,sequence_specification,folding_specification,loop_specification,
               all_specifiers,
               basic_args_to_call_append):
    will_use_registers(circuit,all_registers)


    # 0) Ad hoc initialisation

    #initialise loop_length=1 because:
    # - then too small hairpin loop_length will be loop_length > 4 == 1+
    min_length_hairpin_loop so it is easy to check there <**>
    # - when doing it at the beginning, it is done in parallele whith other initialisation
    circuit.x(loop_length[0])

    #initialise energy to (length-1-min_length_hairpin_loop)*multiloop_flat_contribution
    #it is used for energy computation there <*****>
    #TODO

    # 1) Superposition at the beginning of the Grover search

    initialise_folding(*basic_args_to_call_append)
    initialise_sequence(*basic_args_to_call_append)


    # 2) Check ) matching

    #check if there is a folding[k_right] == ) non matched with a folding[k_left] == (
    #that would then match with a ( at an imaginary position folding[k_left=-1]
    #the result is in continue_searching_right_matching_left:
    # 1 == continue_searching_right_matching_left if ( == folding[k_left=-1] has not been
    matched
    # 0 == continue_searching_right_matching_left otherwise

    characterise_loop(*basic_args_to_call_append,-1,False,False)

```

```
    #increment incorrect_configuration if necessary
    #(it is necessary to increment since the value of incorrect_configuration is unknown since characterise_loop(uncompute=False) can have changed it)
    circuit.append(incr_incorrect.control(ctrl_state="0"),[continue_searching_right_matching_left]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

    inversed(characterise_loop)(*basic_args_to_call_append,-1,False,True)


    # 3) Check ( matching and add energy of loop

    for k_left in range(length-1-min_length_hairpin_loop):


        # 3.1) Compute parameters

        characterise_loop(*basic_args_to_call_append,k_left)


        # 3.2) Check ( matching

        #increment incorrect_configuration if folding[k_left] == ( has no matching )
        circuit.append(incr_incorrect.control(),[continue_searching_right_matching_left]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #TODO: set nb_bits_incorrect (with respect to the following comment)
        #sum all the incorrect flags continue_searching_right_matching_left (1 per position) into incorrect_folding
        #remark that if the flag continue_searching_right_matching_left[k_left=0] == 0 (at k_left=0 incorrect beeing 0)
        #then there is a ) somewhere
        #so continue_searching_right_matching_left[k_left=somewhere] == 0
        #so incorrect_folding is not incremented at each position
        #so the sum cannot overflow


        #continue_searching_right_matching_left has been used
        #and it can be considered that 0 == continue_searching_right_matching_left
        #(don't care about the energy if 1 == continue_searching_right_matching_left since 0 != incorrect_configuration)
        #so it can be used as an auxillary qbit
        #as far as it is cleaned after use (since inversed(characterise_loop) will be applied at the end)


        # 3.3) Check hairpin steric constraint

        #<**> too small hairpin loop_length are loop_length > 4 == 1+min_length_hairpin_loop
        #and 2 < nb_bits_position since 2+min_length_hairpin_loop == 5 <= length
        circuit.append(incr_incorrect.control((nb_bits_position-2)+nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state("(")+n_to_ctrl_state(0,nb_bits_position-2)),[loop_length[i] for i in range(2,nb_bits_position)]+[folding[k_left][i] for i in range(nb_bits_bracket_dot)]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #set loop_length to its normal value
        #in particular it makes easy to check that a loop_length is small enough for the strain_contribution to apply to certain multiloop (there <***>)
        circuit.append(decr_position,[loop_length[i] for i in range(nb_bits_position)])


```

```
        # 3.4) Compute and add energy of loop to energy

        add_loop_energy(*basic_args_to_call_append,k_left)


        # 3.4) Restore parameter qbits

        inversed(characterise_loop)(*basic_args_to_call_append,k_left)

    for k_left in range(length-1-min_length_hairpin_loop,length):
        # 3.2 again) Check ( matching
        circuit.append(incr_incorrect.control(nb_bits_bracket_dot,ctrl_state=
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

    #OPTIMISATION
    #use one less bit for folding[length-1] just by considering ./)
    #use one less bit for folding[0] just by considering ./(


    # 4) Check <-.-> choice

    #|.->. and .<-.| are prohibited
    for k_dot in range(1,length-1):
        #if 0 == folding[k_dot][0] it is a .
        #then
        # - if 1 == folding[k_dot][1] it is a .->
        #   then
        #   (a) 1 == folding[k_dot-1][0] it is a |
        #   and
        #   (b) 0 == folding[k_dot+1][0] it is a .
        #   is prohibited
        #   so by flipping (a) and (b) if 1 == folding[k_dot][1]
        #   then one get
        #   (flipped a) 0 == folding[k_dot-1][0]
        #   and
        #   (flipped b) 1 == folding[k_dot+1][0]
        #   is prohibited
        #   that is the same condition as the other case since
        #   (flipped a) == (c)
        #   and
        #   (flipped b) == (d)
        #
        # - if 0 == folding[k_dot][1] it is a <-.
        #   then
        #   (c) 0 == folding[k_dot-1][0] it is a .
        #   and
        #   (d) 1 == folding[k_dot+1][0] it is a |
        #   is prohibited

        #flip
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])

        #test
        circuit.append(incr_incorrect.control(3*(nb_bits_bracket_dot-1),ctrl_state=
    bracket_dot_to_ctrl_state("|")+bracket_dot_to_ctrl_state(".")+bracket_dot_to_ctrl_state(
    ".")),[folding[k_dot-1][0]]+[folding[k_dot][0]]+[folding[k_dot+1][0]]+[
    incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #flip back
```

```
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])


    # 5) substract the energy of the specification_folding

    #TODO: using loop_specification (which should probably be modified a bit)

    for k_left in range(length-1-min_length_hairpin_loop):
        if "(" == folding_specification[k_left]:
            k_right = loop_specification[k_left][0][3]
            loop_length_specification = loop_specification[k_left][0][3]

            #TODO


    return "RNA_design"
```

Listing 1 – The **`RNA_design()`** function produces the oracle quantum circuit. It orchestrates the analysis of **`folding`** part and the energy computation part. Some elements about the energy computation are still to be done.

2.2.3 Travel the sequence

In order to check if a **`(`** has a matching **`)`** downstream, the automaton strategy is adopted. The register **`count`** stores how many **`(`** are still to be closed. Said differently, it stores the current parenthesis level (counted from the starting point **`k`**). When **`count`** reaches 0, the matching **`)`** has been found.

In order to know whether the loop is a hairpin loop, a multiloop or else, **`nb_enclosed`** counts the number of base pairs at level 1 enclosed in the loop.

The **`loop_length`** is the number of **`.`** at level 1. If the loop is not a multiloop nor a hairpin loop, **`loop_length_righ`** should count the number of **`.`** at level 1 in the loop starting when **`nb_enclosed`** becomes strictly positive.

In case of interior loop (and bulge loop) it is important for the energy computation to copy the appropriate information (information that is not available at the starting point) about the neighbouring of the base pair closing the loop and the base pair enclosed in the loop. That is the role of (in order of the sequence) **`next_left_prev`**, **`next_left`**, **`right_matching_next_left_next`**, **`right_matching_left_prev`** and **`right_matching_left_next`**. And about the choice of **`<`** and **`>`** : **`use_right_matching_left_prev`** and **`use_right_matching_left_next`**.

Handling all these registers at each step of the automaton is the role of the central **`automaton_step()`** function. The role of the **`characterise_loop()`** function is simply to apply **`automaton_step()`** at each position. See Listing 2.

```
def automaton_step(circuit,
                   folding,sequence,count,continue_searching_right_matching_left,nb_enclosed
    ,loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,
    right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,
    use_right_matching_left_next,incorrect_configuration,energy,
                   all_registers,
                   nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                   all_nb_bits,
                   incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,
    incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                   all_gates,
                   length,sequence_specification,folding_specification,loop_specification,
                   all_specifiers,
                   basic_args_to_call_append,
                   k_left,k,
                   compute_energy=True,
```

```
                    uncompute=False):
    if compute_energy:
        will_use_registers(circuit,[folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next])
        if not uncompute:
            will_use_registers(circuit,[incorrect_configuration])
    else:
        will_use_registers(circuit,[folding,count,continue_searching_right_matching_left])

    #<*> if it has been checked that there is a matching ) to find,
    #then continue_searching_right_matching_left == 1
    #otherwise continue_searching_right_matching_left == 0


    #1), 2) and 3) before 4) Update count
    #and
    #5) and 6) after 4) Update count
    #so that things are always controled by 1 == count
    #(which could be used by the compiler for optimisation when implementing sevral time in
    a row the same MCXGate())

    if compute_energy:


        # 1) Get next_left_prev and next_left

        for i in range(nb_bits_base):
            #if 0 < k (which is always the case when compute_energy)
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="
    1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state("(")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k-1][i],next_left_prev[i]])
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="
    1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state("(")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k  ][i],next_left     [i]])


        # 2) Get right_matching_left_prev and right_matching_left_next

        #get right_matching_left_prev and use_right_matching_left_prev
        for i in range(nb_bits_base):
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
         ,ctrl_state="1"                                +n_to_ctrl_state(1,nb_bits_position)+
    bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j]
    for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[
    sequence[k-1][i],right_matching_left_prev[i]])
            circuit.append(        MCXGate(1+nb_bits_bracket_dot+nb_bits_position+
    nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state(".->")+n_to_ctrl_state(1,
    nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)]+[
    folding[k][i] for i in range(nb_bits_bracket_dot)]+[folding[ k-1][i] for i in range(
    nb_bits_bracket_dot)]+[use_right_matching_left_prev])

        #FUTURE: if one adds GU pairs, then one base of a valid base pair would not be
    enough anymore to know which base pair it is
        #so there would be a need for right_matching_left, this way:
        #for i in range(nb_bits_base):
        #    circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
```

```
     ,ctrl_state="1"                                  +n_to_ctrl_state(1,nb_bits_position)+
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j]
for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[
sequence[k  ][i],right_matching_left[i]     ])

        #get right_matching_left_next and use_right_matching_left_next
        if length-1 > k:
            for i in range(nb_bits_base):
                circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
     ,ctrl_state="1"                                  +n_to_ctrl_state(1,nb_bits_position)+
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j]
for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[
sequence[k+1][i],right_matching_left_next[i]])
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+
nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state("<-.")+n_to_ctrl_state(1,
nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)]+[
folding[k][i] for i in range(nb_bits_bracket_dot)]+[folding[ k+1][i] for i in range(
nb_bits_bracket_dot)]+[use_right_matching_left_next])
        #else use_right_matching_left_next=0


        # 3) Check base pairing rules

        #FUTURE: if one adds GU pairs, then there would be a right_matching_left
        #so base pairing rules could be checked more efficiently in characterise_loop()

        #the base encoding is such that cx(sequence[k_left],sequence[k_right]) leads to
        #"11" == sequence[k_right] if and only if (sequence[k_left],sequence[k_right]) is a
valid Watson-Crick base pair
        for i in range(nb_bits_base):
            circuit.cx(sequence[k_left][i],sequence[k][i])
        if not uncompute:
            circuit.append(incr_incorrect.control(1+nb_bits_bracket_dot+nb_bits_position+
nb_bits_base,ctrl_state="11"+n_to_ctrl_state(1,nb_bits_position)+
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[i]
for i in range(nb_bits_position)]+[folding[k][i] for i in range(nb_bits_bracket_dot)]+[
sequence[k][i] for i in range(nb_bits_base)]+[incorrect_configuration[i] for i in range(
nb_bits_incorrect)])
        #there is actually no need to restore sequence[k]


    # 4) Update count

    #if 1 == continue_searching_right_matching_left
    #then update count for the next candidate
    #OPTIMISATION: use smaller increment and decrement circuits when the maximum value of
count enables it (since it never goes negative)
    if 0 == k:
        #then -1 == k_left and ( == folding[k=-1]
        # - so 1 == continue_searching_right_matching_left
        #   so it is ok not controlling by continue_searching_right_matching_left here
        # - so 1 == count
        #   so there is no need to incr_position or decr_position in order to increment or
decrement
        circuit.append(MCXGate((nb_bits_bracket_dot-1),ctrl_state=bracket_dot_to_ctrl_state(
"|")),[folding[k][0]]+[count[0]])
        circuit.append(MCXGate( nb_bits_bracket_dot   ,ctrl_state=bracket_dot_to_ctrl_state(
"(")),[folding[k][i] for i in range(nb_bits_bracket_dot)]+[count[1]])
    else:
        circuit.append(incr_position.control(1+nb_bits_bracket_dot,ctrl_state=
```

```
    bracket_dot_to_ctrl_state("(")+"1"),[continue_searching_right_matching_left]+[folding[k
    ][i] for i in range(nb_bits_bracket_dot)]+[count[i] for i in range(nb_bits_position)])
92      circuit.append(decr_position.control(1+nb_bits_bracket_dot,ctrl_state=
    bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[folding[k
    ][i] for i in range(nb_bits_bracket_dot)]+[count[i] for i in range(nb_bits_position)])
93
94  #if 0 == continue_searching_right_matching_left then simply increment count so that it
    never equals 0 again
95  circuit.append(incr_position.control(ctrl_state="0"),[
    continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)])
96
97
98  if compute_energy:
99
100
101     # 5) Get right_matching_next_left_next
102
103     if k < length-1:
104         for i in range(nb_bits_base):
105             #FUTURE: if one adds GU pairs, then one base of a valid base pair is not
    enough anymore to know which base pair it is
106             #so there would be a need for right_matching_next_left, this way:
107             #circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,"1"+
    n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k  ][i],
    right_matching_next_left[i]     ])
108             circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="
    1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k+1][i],
    right_matching_next_left_next[i]])
109
110
111     # 6) Update nb_enclosed
112
113     if 0 < nb_bits_enclosed:
114         #OPTIMISATION: use smaller increment circuit when the maximum value of
    nb_enclose enables it (which depends on min_length_hairpin_loop)
115         circuit.append(incr_nb_enclosed.control(1+nb_bits_bracket_dot+nb_bits_position,
    ctrl_state=n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)])
116
117
118     # 7) Update loop_length
119
120     #loop_length is the number of . for which 1 == count
121
122     #if 1 == continue_searching_right_matching_left and 1 == count and "x0" == . ==
    folding[k]
123     #then increment loop_length
124     #OPTIMISATION: use smaller increment circuit when the maximum value of loop_length
    enables it
125     if k_left+1 == k:
126         #since the initial value 1 == loop_length is known then there is no need to
    incr_position in order to increment
127         #two CXGate() are sufficient
128         circuit.append(MCXGate(              (nb_bits_bracket_dot-1)+nb_bits_position+1,
    ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
```

```
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left,loop_length[0]])
129             circuit.cx(loop_length[0],loop_length[1],ctrl_state="0")
130         else:
131             circuit.append(incr_position.control((nb_bits_bracket_dot-1)+nb_bits_position+1,
    ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left]+[loop_length[i] for i in range(nb_bits_position)
    ])
132
133
134         # 8) Update loop_length_right
135
136         #in case of a non multi-loop,
137         #(that is 1 => nb_enclosed)
138         #loop_length_right is the number of . for which 1 == count after the first enclosed
    stem (if any)
139
140         #if 1 == continue_searching_right_matching_left and 1 == count and "x0" == . ==
    folding[k]
141         #and 1 == nb_enclosed (that is 1 == nb_enclosed[0] since 1 => nb_enclosed)
142         #then increment loop_length_right
143         #OPTIMISATION: use smaller increment circuit when the maximum possible value of
    loop_length enables it
144         if k_left+2+min_length_hairpin_loop < k and 0 < nb_bits_enclosed:
145             circuit.append(incr_position.control((nb_bits_bracket_dot-1)+nb_bits_position+2,
    ctrl_state="11"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left,nb_enclosed[0]]+[loop_length_right[i] for i in
    range(nb_bits_position)])
146
147
148     # 9) Update continue_searching_right_matching_left
149
150     #if 0 == count
151     #then there is no more matching ) to find
152     circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(0,nb_bits_position))
        ,[count[i] for i in range(nb_bits_position)]+[continue_searching_right_matching_left])
153
154     return "step"
```

Listing 2 – The **`automaton_step()`** function is the algorithmically complicated function. It is also quite modular and by modifying it, one could put different information at **`compute_energy()`**'s disposal, thus adapting the algorithm to another RNA free energy computation model.

2.2.4 Computation of energy

Even though the computation of energy part is quite long (and not finished yet), it is quite straightforward since the algorithmic work has been isolated and done in the **`automaton_step()`** function.

It is basically a conditional sum of parameters guided by the information provided by the **`automaton_step()`** function. The typical structure is exemplified in point « 2.2) GG interior mismatch » (see Appendix A).

2.3 Simulation

The quantum circuits are simulated using a statevector simulator, that is a simulator that do the matrix computations corresponding to the quantum gates and do not simulate any hardware noise.

3 Results

Since the programming is not finished yet, the only results are partial tests. We aim to know if the **`characterise_loop()`** function works as expected since it is the algorithmic core of the project and it is entirely coded.

In order to test a function that produces a quantum circuit, we should simulate that quantum circuit. However, even for the smallest case (RNA of length 5) is not possible to simulate the entire circuit since it needs too many qbits.

Instead, we simulate a simplified version :

— we ignore the parts supposed to detect, compute and store information for the computation of energy,

— we keep all about recognising a customised dot-bracket word describing a fully specifies RNA secondary structure (except the test of the size of hairpin loops based on the register **`length`**).

We test it for 5-base-long RNA. As a result we obtain customised dot-bracket word as well as values of the register **`wrong`** and other registers.

When **`wrong`** = 0, customised dot-bracket words are well parenthesised with no **`.<|`** pattern nor **`|>.`** pattern. Among such words, the only one missing are those with **`(`** at a too small distance from the end to form a hairpin loop of size at least 3.

Other customised dot-bracket words appear only when **`wrong`** > 0.

Interpretation The test of the size of hairpin loop is optimised near the end and does not use the length of the loop (computed in the register **`length`**). So the results are as expected : **`characterise_loop()`** is able to detect customised dot-bracket words describing a fully specifies RNA secondary structure for 5-base-long RNA.

4 Discussion

The presented work is not completed. However, a progress report can be set on the light of these preliminary results.

4.1 Simplification of a realistic model

The free energy computation model used here is a simplification of the nearest neighbour model [TM10] that is a reference biologically realistic model of RNA secondary structure. It is quite a realistic model compared to other quantum approaches of the (inverse) folding problem of RNA (see for example [Zab+22] where the free energy of a configuration only depends on the secondary structure and not on the sequence). Moreover, the simplifications adopted here are careful of the biological relevance of the free energy model. The less satisfying simplification is the ignorance of pseudoknots.

These simplifications enabled to :

— get the programming go fast enough to reach the first test phase (the test of the analysis of the candidate customised dot-bracket word),

— get some features testable without any specialised computing resources, because the number of qbits is small enough for small cases thanks to simplifications.

4.2 Tackle the computational difficulties

Although we ignore the coaxial stacking, we handle the non local mechanism of dandling end and terminal mismatch choice, that is computationally similar. That kind of global free energy minimisation is the kind of phenomenon that make the folding problem of RNA and the inverse folding problem of RNA so difficult to solve.

This simplification only aims at reducing the number of qbits (see Remark 2) and do not ignore the computational difficulty.

4.3 The presented algorithm can easily be generalised

The automaton approach chosen here is quite flexible and can be adapted to detecting other patterns with specific free energy contribution, the first of which being GU pairs. Moreover, the separation of the free energy computation and the analysis of the candidate secondary structure results in a modularity that favour a more involved transformation of the analysis of secondary structure part. That could lead to take pseudoknots into account.

But before that, the programming of algorithm described here should be completed, paving the way to a first working version and a potential publication of this work.

Références

[Bro20] Jessica A Brown. « Unraveling the structure and biological functions of RNA triple helices ». en. In : *Wiley Interdiscip. Rev. RNA* 11.6 (nov. 2020), e1598.

[TRG21] Martina Tassinari, Sara N Richter et Paolo Gandellini. « Biological relevance and therapeutic potential of G-quadruplex structures in the human noncoding transcriptome ». In : *Nucleic Acids Research* 49.7 (mars 2021), p. 3617-3633. issn : 0305-1048. doi : 10.1093/nar/gkab127. eprint : https://academic.oup.com/nar/article-pdf/49/7/3617/37123751/gkab127.pdf. url : https://doi.org/10.1093/nar/gkab127.

[Hof+98] Ivo Hofacker et al. « Fast Folding and Comparison of RNA Secondary Structures (The Vienna RNA Package) ». In : *Monatsh Chem* 125 (oct. 1998).

[MWS16] Maureen McKeague, Remus S Wong et Christina D Smolke. « Opportunities in the design and application of RNA for gene expression control ». en. In : *Nucleic Acids Res.* 44.7 (avr. 2016), p. 2987-2999.

[TM10] Douglas H Turner et David H Mathews. « NNDB : the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure ». en. In : *Nucleic Acids Res.* 38.Database issue (jan. 2010), p. D280-2.

[Kha+23] Mohammad Hassan Khatami et al. « Gate-based quantum computing for protein design ». en. In : *PLoS Comput. Biol.* 19.4 (avr. 2023), e1011033.

[Chu+18] Alexander Churkin et al. « Design of RNAs : comparing programs for inverse RNA folding ». en. In : *Brief. Bioinform.* 19.2 (mars 2018), p. 350-358.

[Lor+11] Ronny Lorenz et al. « ViennaRNA Package 2.0 ». en. In : *Algorithms Mol. Biol.* 6.1 (nov. 2011), p. 26.

[Gro96] Lov K. Grover. « A Fast Quantum Mechanical Algorithm for Database Search ». In : *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA : Association for Computing Machinery, 1996, p. 212-219. isbn : 0897917855. doi : 10.1145/237814.237866. url : https://doi.org/10.1145/237814.237866.

[DH99] Christoph Durr et Peter Hoyer. *A Quantum Algorithm for Finding the Minimum*. 1999. arXiv : quant-ph/9607014 [quant-ph].

[Zab+22] Tristan Zaborniak et al. « A QUBO model of the RNA folding problem optimized by variational hybrid quantum annealing ». In : *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, sept. 2022. doi : 10.1109/qce53715.2022.00037. url : https://doi.org/10.1109%2Fqce53715.2022.00037.

[Cuc+04] Steven A. Cuccaro et al. *A new quantum ripple-carry addition circuit*. 2004. arXiv : quant-ph/0410184 [quant-ph].

A Raw code

Code is given in the order of expected execution.

A.1 Oracle's code

```
import numpy as np
from math import pi,log2,floor,ceil
from qiskit import *
from qiskit.circuit import *
from qiskit.extensions import *
from qiskit.circuit.library import *
from qiskit.extensions.simulator.snapshot import snapshot
from qiskit.quantum_info.operators import Operator
from qiskit.extensions.simulator.snapshot import snapshot
from scipy import optimize
from matplotlib.pyplot import plot,show
%matplotlib inline
%config InlineBackend.figure_format = 'svg' # Makes the images look nice
```

Listing 3 – Requirements

```
#MAJ
q=QuantumRegister(3)
MAJ = QuantumCircuit(q)

MAJ.cnot(q[2],q[1])
MAJ.cnot(q[2],q[0])
MAJ.ccx(q[0],q[1],q[2])

maj = MAJ.to_gate(label='MAJ')

#UMA
q=QuantumRegister(3)
UMA = QuantumCircuit(q)

UMA.ccx(q[0],q[1],q[2])
UMA.cnot(q[2],q[0])
UMA.cnot(q[0],q[1])

uma = UMA.to_gate(label='UMA')

#add
def add_circuit(nb_bits):
    concerved = QuantumRegister(nb_bits, name="=")
    not_concerved = QuantumRegister(nb_bits, name="->")
    initial_carry = QuantumRegister(1)
    add = QuantumCircuit(initial_carry,concerved,not_concerved)
    add.append(maj,[initial_carry,not_concerved[0],concerved[0]])
    for i in range(1,nb_bits):
        add.append(maj,[concerved[i-1],not_concerved[i],concerved[i]])
    for i in range(nb_bits-1,0,-1):
        add.append(uma,[concerved[i-1],not_concerved[i],concerved[i]])
    add.append(uma,[initial_carry,not_concerved[0],concerved[0]])
    return add

def add_gate(nb_bits):
    return add_circuit(nb_bits).to_gate(label="add{}".format(nb_bits))

```

```
add_circuit(6).draw()
```

Listing 4 – Adder mod 2^n [Cuc+04].

```
def incr_circuit(nb_bits):
    number = QuantumRegister(nb_bits,name="number")
    incr = QuantumCircuit(number)
    for i in range(nb_bits-1):
        incr.append(MCXGate(nb_bits-i-1),[number[j] for j in range(nb_bits-i)])
    incr.x(number[0])
    return incr

def incr_gate(nb_bits):
    return incr_circuit(nb_bits).to_gate(label=f"incr{nb_bits}")

def decr_circuit(nb_bits):
    return incr_circuit(nb_bits).inverse()

def decr_gate(nb_bits):
    return decr_circuit(nb_bits).to_gate(label=f"decr{nb_bits}")

incr_circuit(6).draw()
decr_circuit(6).draw()
```

Listing 5 – Increment and decrement.

```
#set a register (initially at 0) to a constant
def set_cst_circuit(constant,nb_bits):
    number = QuantumRegister(nb_bits,name="number")
    add_cst = QuantumCircuit(number)

    #TODO

    return add_cst

#add a constant to a register
def add_cst_circuit(constant,nb_bits):
    number = QuantumRegister(nb_bits,name="number")
    add_cst = QuantumCircuit(number)

    #TODO

    return add_cst

def add_cst_gate(constant,nb_bits):
    return add_cst_circuit(constant,nb_bits).to_gate(label=f"(+){constant}")

def substract_cst_circuit(constant,nb_bits):
    return add_cst_circuit(-1*constant,nb_bits).inverse()

def substract_cst_gate(constant,nb_bits):
    return substract_cst_circuit(constant,nb_bits).to_gate(label=f"(-){constant}")
```

Listing 6 – Arithmetics with constants remains to be done.

```
#keywords to find important comments:
#OPTIMISATION: there is (or may be) an optimisation here
#FUTURE: something that should be done if a certain feature was added in the future
#AMPLITUDE: diffrent configurations does not have the same amplitude, so it may be important
    for the Grover search
#GOOD PRACTICE: this is NOT a reliable way to do, please change it
#TODO
```

```
7
8  nb_bits_bracket_dot = 2
9  nb_bits_base = 2
10 nb_bits_loop_type = 2
11 nb_bits_energy = 5 #TODO
12 min_length_hairpin_loop = 3
13
14 def nb_bits_wanna_count_from_zero_until(n):
15     return ceil(log2(float(n)))+1
16
17 def n_to_ctrl_state(n,nb_bits):
18     return format(n,"0{}b".format(nb_bits))[:nb_bits]
19
20 def base_to_ctrl_state(base):
21     if "A" == base:
22         return "00"
23     if "C" == base:
24         return "01"
25     if "G" == base:
26         return "10"
27     if "U" == base:
28         return "11"
29     raise Exception("Not a base.")
30
31
32 def bracket_dot_to_ctrl_state(bd):
33     if "(" == bd:
34         return "01"
35     if ")" == bd:
36         return "11"
37     if "." == bd:
38         return "0"
39     if "|" == bd:
40         return "1"
41     if "<-." == bd:
42         return "00"
43     if ".->" == bd:
44         return "10"
45
46
47 def base_to_set(base):
48     if "N" == base or "." == base:
49         return set(("A","C","G","U"))
50     elif "A" == base:
51         return set(("A"))
52     elif "C" == base:
53         return set(("C"))
54     elif "G" == base:
55         return set(("G"))
56     elif "U" == base:
57         return set(("U"))
58     elif "R" == base:
59         return set(("A","G"))
60     elif "Y" == base:
61         return set(("C","U"))
62     elif "S" == base:
63         return set(("C","G"))
64     elif "W" == base:
65         return set(("A","U"))
66     elif "K" == base:
67         return set(("G","U"))
```

```
    elif "M" == base:
        return set(("A","C"))
    elif "B" == base:
        return set(("C","G","U"))
    elif "D" == base:
        return set(("A","G","U"))
    elif "H" == base:
        return set(("A","C","U"))
    elif "V" == base:
        return set(("A","C","G"))


def base_matching_Watson_Crick(base):
    if "A" == base:
        return "U"
    if "C" == base:
        return "G"
    if "G" == base:
        return "C"
    if "U" == base:
        return "A"
    raise Exception("Not a base.")

def base_pair_complement(base_set):
    complement = set(())
    for base in base_set:
        complement.add(base_matching_Watson_Crick(base))
    return complement

def is_valid_base_pair(left,right):
    left = base_to_set(left)
    right = base_to_set(right)
    right = base_pair_complement(right)
    return not left.intersection(right).issubset(set(()))

def characterise_loop_specification(length,folding_specification,sequence_specification,
    folding_specification_param,sequence_specification_param):
    sequence_specification_param = (sequence_specification if None ==
    sequence_specification_param else sequence_specification_param)

    length = len(folding_specification)
    bottom = [None,None,None,-1,0]
    bottom[0] = bottom
    bottom[1] = bottom
    bottom[2] = bottom
    loop_specification = [[bottom,bottom,bottom,k,0] for k in range(length)]

    #semantics of loop_specification

    #[0] matching
    #if folding_specification: k_left(-k_right)
    #then, compute loop_specification[k_left][0]=loop_specification[k_right]
    #otherwise let loop_specification[k_left][0]=None


    #if folding_specification: k_prev(-k_next(
    #or folding_specification: k_prev)-k_next)

    #[1] next
    #then, compute loop_specification[k_prev][1]=loop_specification[k_next]
    #otherwise let loop_specification[k_prev][1]=None
```

```
    #[2] previous
    #then, compute loop_specification[k_next][2]=loop_specification[k_prev]
    #otherwise let loop_specification[k_next][2]=None

    #[3] position

    #[4] loop length if "(" == folding_specification[k]
    #0 otherwise

    open_loop_stack = []
    prev_open = None
    prev_close = None
    for k in range(length):
        if "(" == folding_specification[k]:
            if None != prev_open:
                prev_open[1] = loop_specification[k]
                loop_specification[k][2] = prev_open
            open_loop_stack.append(loop_specification[k])
            prev_open = loop_specification[k]
            prev_close = None
        elif ")" == folding_specification[k]:
            if None != prev_close:
                prev_close[1] = loop_specification[k]
                loop_specification[k][2] = prev_close
            try:
                left = open_loop_stack.pop()
            except IndexError:
                raise Exception(f"Wrong folding specification format: \"{
folding_specification_param[k]}\" in folding_specification at index {k} has no matching
\"(\".")
            left[0] = loop_specification[k]
            if not is_valid_base_pair(sequence_specification[left[3]],sequence_specification
[k]):
                raise Exception(f"Folding specification and sequence specification do not
match: "+
                                f"\"{folding_specification_param[left[3]]}\" in
folding_specification at index {left[3]} "+
                                f"matches "+
                                f"\"{folding_specification_param[k      ]}\" in
folding_specification at index {k}, "+
                                f"but "+
                                f"\"{sequence_specification_param[left[3]]}\" in
sequence_specification at index {left[3]} "+
                                f"does not form a valid base pair with "+
                                f"\"{sequence_specification_param[k      ]}\" in
sequence_specification at index {k}.")
            prev_close = loop_specification[k]
            prev_open = None
        elif "." == folding_specification[k]:
            if 0 < len(open_loop_stack):
                open_loop_stack[-1][4]+=1
        else:
            raise Exception(f"Wrong folding specification format: \"{
folding_specification_param[k]}\" found in folding_specification at index {k}.")
    if 0 < len(open_loop_stack):
        raise Exception(f"Wrong folding specification format: \"{folding_specification_param
[open_loop_stack[-1][3]]}\" in folding_specification at index {k} has no matching \")\".
")
    return loop_specification

```

```
177 def will_use_registers(circuit,registers):
178     for reg in registers:
179         if list != type(reg):
180             if not circuit.has_register(reg):
181                 circuit.add_register(reg)
182         else:
183             for r in reg:
184                 if not circuit.has_register(r):
185                     circuit.add_register(r)
186 
187 
188 #GOOD PRACTICE: please use classes instead of this!
189 
190 def to_circuit_builder(append):
191     def circuit_builder(folding_specification,sequence_specification=None,*other_args):
192         #save parameters for error messages
193         folding_specification_param = folding_specification
194         sequence_specification_param = sequence_specification
195 
196         #semantics of sequence_specification
197         #follow the IUPAC code
198         #R == A or G
199         #Y == C or U
200         #S == G or C
201         #W == A or U
202         #K == G or U
203         #M == A or C
204         #B == C or G or U
205         #D == A or G or U
206         #H == A or C or U
207         #V == A or C or G
208         #N == A or C or G or U
209 
210         #check the input is well formed
211 
212         folding_specification = str(folding_specification)
213         length = len(folding_specification)
214 
215         if length < min_length_hairpin_loop+2:
216             raise Exception(f"folding_specification's length is to small (minimum size is {
    min_length_hairpin_loop+2}).")
217 
218         if None == sequence_specification:
219             sequence_specification="N"*length
220         else:
221             sequence_specification=str(sequence_specification).upper()
222             if length != len(sequence_specification):
223                 raise Exception(f"Wrong sequence specification format:
    sequence_specification's length ({len(sequence_specification_param)}) is different from
    folding_specification's length ({len(folding_specification_param)}).")
224 
225         loop_specification = characterise_loop_specification(length,folding_specification,
    sequence_specification,folding_specification_param,sequence_specification_param)
226 
227         #TODO: check that characterise_loop_specification() works well
228  #       for a in loop_specification:
229  #           print(a)
230 
231         all_specifiers = [length,sequence_specification,folding_specification,
    loop_specification]
232 
```

```
        nb_bits_position = nb_bits_wanna_count_from_zero_until(length) #from 0 until length because we add a ( at the begining to check the folding is well formed
        max_enclosed_loop = (length-2)//(2+min_length_hairpin_loop)
        nb_bits_enclosed = 0 if max_enclosed_loop == 0 else nb_bits_wanna_count_from_zero_until(max_enclosed_loop) #from 0 until the maximum number of loops (of folding (...)) that could be enclosed in a multiloop
        #TODO set nb_bits_incorrect
        #nb_bits_incorrect = nb_bits_wanna_count_from_zero_until(length+length)#from 0 because 0==incorrect_configuration when the configuration is correct, length (and not length+1 indeed) is the maximum number of incrementations of incorrect_configuration due to incorrect folding and length is the maximum contribution of incorrect base pairing to incorrect_configuration
        nb_bits_incorrect = 6 #enough for what can be tested on a personnal computer

        all_nb_bits=[nb_bits_position,nb_bits_enclosed,nb_bits_incorrect]

        incr_position=incr_gate(nb_bits_position)
        decr_position=decr_gate(nb_bits_position)
        add_position=add_gate(nb_bits_position)
        incr_incorrect=incr_gate(nb_bits_incorrect)
        decr_incorrect=decr_gate(nb_bits_incorrect)
        incr_nb_enclosed= (IGate() if 0 == nb_bits_enclosed else incr_gate(nb_bits_enclosed))
        incr_energy=incr_gate(nb_bits_energy)
        decr_energy=decr_gate(nb_bits_energy)
        add_energy=add_gate(nb_bits_energy)

        all_gates=[incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy]

        #input
        #semantics of folding
        #<-. == "00"
        #.-> == "10"
        #) == "11"
        #( == "01"
        #so
        #. == "x0"
        #| == "x1"
        folding = [QuantumRegister(nb_bits_bracket_dot,name=f"(/./h/){k}") for k in range(length)]
        #semantics of sequence
        #A == "00"
        #C == "01"
        #G == "10"
        #U == "11"
        sequence = [QuantumRegister(nb_bits_base,name=f"A/U/G/C{k}") for k in range(length)]

        #auxillary qbits
        count = QuantumRegister(nb_bits_position,name="count") #count how many parenthesis still need to be closed
        continue_searching_right_matching_left = QuantumRegister(1,name="continue_match")
        nb_enclosed = QuantumRegister(nb_bits_enclosed,name="enclosed") #count how many stems are enclosed in the loop
        loop_length = QuantumRegister(nb_bits_position,name="length") #loop_length is initialised to 1 (actually not 0 because of an optimisation used there <**>) and is incremented each time 1 == count and . == folding until 0 == count
        loop_length_right = QuantumRegister(nb_bits_position,name="length_right")
        #in the order of the sequence:
        next_left_prev = QuantumRegister(nb_bits_base,name="next_left_prev")
        next_left = QuantumRegister(nb_bits_base,name="next_left")
```

```
        right_matching_next_left_next = QuantumRegister(nb_bits_base,name="next_right_next")
        right_matching_left_prev = QuantumRegister(nb_bits_base,name="right_prev")
        right_matching_left_next = QuantumRegister(nb_bits_base,name="right_next")
        use_right_matching_left_prev = QuantumRegister(1,name="right_prev?")
        use_right_matching_left_next = QuantumRegister(1,name="right_next?")

        #results
        incorrect_configuration = QuantumRegister(nb_bits_incorrect,name="incorrect")
        energy = QuantumRegister(nb_bits_energy,name="energy_delta")

        all_registers=[folding,sequence,count,continue_searching_right_matching_left,
    nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,
    right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,
    use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy
    ]

        circuit = QuantumCircuit()

        def semantics(bit_string):
            i_bit = 0
            semantics_string = ""

            def semantics_number(label,nb_bits_number):
                s=str(int(bit_string[i_bit:i_bit+nb_bits_number],2))
                delta=nb_bits_number-len(s)
                return s+"_"+label+delta*" "+" ",nb_bits_number

            def semantics_base(label):
                s=("A" if bit_string[i_bit:i_bit+nb_bits_base]==base_to_ctrl_state("A") else
                   "C" if bit_string[i_bit:i_bit+nb_bits_base]==base_to_ctrl_state("C") else
                   "G" if bit_string[i_bit:i_bit+nb_bits_base]==base_to_ctrl_state("G") else
                   "U" if bit_string[i_bit:i_bit+nb_bits_base]==base_to_ctrl_state("U") else
     "")
                return s+"_"+label+" ",nb_bits_base

            if circuit.has_register(energy):
                s,i=semantics_number("E",nb_bits_energy)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(incorrect_configuration):
                s,i=semantics_number("w",nb_bits_incorrect)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(use_right_matching_left_next):
                s,i=semantics_number("Mn?",1)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(use_right_matching_left_prev):
                s,i=semantics_number("Mp?",1)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(right_matching_left_next):
                s,i=semantics_base("Mn")
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(right_matching_left_prev):
                s,i=semantics_base("Mp")
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(right_matching_next_left_next):
                s,i=semantics_base("Rn")
```

```
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(next_left):
                s,i=semantics_base("L")
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(next_left_prev):
                s,i=semantics_base("Lp")
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(loop_length_right):
                s,i=semantics_number("n2",nb_bits_position)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(loop_length):
                s,i=semantics_number("n",nb_bits_position)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(nb_enclosed):
                s,i=semantics_number("e",nb_bits_enclosed)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(continue_searching_right_matching_left):
                s,i=semantics_number("s",1)
                semantics_string+=s
                i_bit+=i
            if circuit.has_register(count):
                s,i=semantics_number("c",nb_bits_position)
                semantics_string+=s
                i_bit+=i

            if circuit.has_register(sequence[0]):
                semantics_string+="\n"
                for k in range(length-1,-1,-1):
                    semantics_string+=("A" if bit_string[i_bit+k*nb_bits_base:i_bit+(k+1)*
    nb_bits_base]==base_to_ctrl_state("A") else
                                       "C" if bit_string[i_bit+k*nb_bits_base:i_bit+(k+1)*
    nb_bits_base]==base_to_ctrl_state("C") else
                                       "G" if bit_string[i_bit+k*nb_bits_base:i_bit+(k+1)*
    nb_bits_base]==base_to_ctrl_state("G") else
                                       "U" if bit_string[i_bit+k*nb_bits_base:i_bit+(k+1)*
    nb_bits_base]==base_to_ctrl_state("U") else "")
                i_bit+=length*nb_bits_base
            if circuit.has_register(folding[0]):
                semantics_string+="\n"
                for k in range(length-1,-1,-1):
                    semantics_string+=("<" if bit_string[i_bit+k*nb_bits_bracket_dot:i_bit+(
    k+1)*nb_bits_bracket_dot]==bracket_dot_to_ctrl_state("<-.") else
                                       ">" if bit_string[i_bit+k*nb_bits_bracket_dot:i_bit+(
    k+1)*nb_bits_bracket_dot]==bracket_dot_to_ctrl_state(".->") else
                                       "(" if bit_string[i_bit+k*nb_bits_bracket_dot:i_bit+(
    k+1)*nb_bits_bracket_dot]==bracket_dot_to_ctrl_state("(")   else
                                       ")" if bit_string[i_bit+k*nb_bits_bracket_dot:i_bit+(
    k+1)*nb_bits_bracket_dot]==bracket_dot_to_ctrl_state(")")   else "")
                i_bit+=length*nb_bits_bracket_dot
            return semantics_string

        basic_args_to_call_append=[circuit]+all_registers+[all_registers]+all_nb_bits+[
    all_nb_bits]+all_gates+[all_gates]+all_specifiers+[all_specifiers]
        basic_args_to_call_append.append(basic_args_to_call_append)

```

```
        label=append(*basic_args_to_call_append,*other_args)
        return circuit,semantics,label
    return circuit_builder

def to_gate_builder(append):
    def gate_builder(length,*other_args):
        circ,_,lab=to_circuit_builder(append)(length,sequence_specification,
    folding_specification,
                                              *other_args)
        return circ.to_gate(label=lab)
    return gate_builder

def to_gate_inverse_builder(append):
    def gate_inverse_builder(length,sequence_specification,folding_specification,
                             *other_args):
        circ,_,lab=to_circuit_builder(append)(length,sequence_specification,
    folding_specification,
                                              *other_args)
        return circ.inverse().to_gate(label=lab+"^-1"),
    return gate_inverse_builder

def inversed(append):
    def append_inversed(circuit,
                        folding,sequence,count,continue_searching_right_matching_left,
    nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,
    right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,
    use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy
    ,
                        all_registers,
                        nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                        all_nb_bits,
                        incr_position,decr_position,add_position,incr_incorrect,
    decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                        all_gates,
                        length,sequence_specification,folding_specification,
    loop_specification,
                        all_specifiers,
                        basic_args_to_call_append,
                        *other_args):
        circ = QuantumCircuit()
        #add QuantumRegisters to the circ and append gates
        basic_args_to_call_append[0]=circ
        lab=append(*basic_args_to_call_append,*other_args)
        basic_args_to_call_append[0]=circuit
        all_registers_list = folding+sequence+[all_registers[i] for i in range(2,len(
    all_registers))]
        register_list =list(np.asarray(all_registers_list,dtype=Register)[[circ.has_register
    (reg) for reg in all_registers_list]])

        will_use_registers(circuit,register_list)
        qbit_list = [qbit for register in register_list for qbit in register]
        circuit.compose(circ.inverse(),qbit_list,inplace=True)
        return lab+"^-1"
    return append_inversed
```

Listing 7 – Input management, and programming tools that aim to simplify the rest. (No scientific interest.)

```
#TODO: extract energy parameters directly from the viennaRNA package for modularity and
    maintenance
#instead of (simingly 2011-old) tables from the NNDB (Nearest Neighbour model Data Base)
#here https://rna.urmc.rochester.edu/NNDB/turner04/index.html

```

```

#TODO: energy parameter conventions

AU_end_penalty = 0.45
helix_flat_contribution = 4.09
special_C_bulge_bonus = -0.9
hairpin_mismatch_UU_AG_contribution = -0.9
hairpin_mismatch_GG_contribution = -0.8
hairpin_C3_contribution = 1.5

#multiloop coefficients
multiloop_flat_contribution = 9.25#    0.91
one_branch_contribution = -0.63#    0.24
strain_contribution = 3.14#    0.44
```

Listing 8 – Energy computation parameter management (to be done).

```
def mismatch_contribution(base_mismatch5,base_pair5,base_mismatch3):
    return [
        #"A" == base_left
        [[-1.0, -0.8, -1.1, -0.8],
         [-0.7, -0.6, -0.7, -0.5],
         [-1.1, -0.8, -1.2, -0.8],
         [-0.7, -0.6, -0.7, -0.5]],

        #"C" == base_left
        [[-1.1, -1.5, -1.3, -1.5],
         [-1.1, -0.7, -1.1, -0.5],
         [-1.6, -1.5, -1.4, -1.5],
         [-1.1, -1.0, -1.1, -0.7]],

        #"G" == base_left
        [[-1.5, -1.5, -1.4, -1.5],
         [-1.0, -1.1, -1.0, -0.8],
         [-1.4, -1.5, -1.6, -1.5],
         [-1.0, -1.4, -1.0, -1.2]],

        #"U" == base_left
        [[-0.8, -1.0, -0.8, -1.0],
         [-0.6, -0.7, -0.6, -0.7],
         [-0.8, -1.0, -0.8, -1.0],
         [-0.6, -0.8, -0.6, -0.8]]
    ][int(base_to_ctrl_state(base_mismatch5),2)][int(base_to_ctrl_state(base_pair5),2)][int(
    base_to_ctrl_state(base_mismatch3),2)]


def substract_enclosed_mismatch_contribution(circuit,
                                             folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
    incorrect_configuration,energy,
                                             all_registers,
                                             nb_bits_position,nb_bits_enclosed,
    nb_bits_incorrect,
                                             all_nb_bits,
                                             incr_position,decr_position,add_position,
    incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                                             all_gates,
                                             length,sequence_specification,
    folding_specification,loop_specification,
                                             all_specifiers,
```

```
                                            basic_args_to_call_append,
                                            k_left,
                                            base_next_left_prev,base_next_left,
    base_right_matching_next_left_next):
    #parameters base_ are in the order of the sequence
    will_use_registers(circuit,[folding,sequence,continue_searching_right_matching_left,
    right_matching_left_next,use_right_matching_left_next,energy])
    should_not_cancel = continue_searching_right_matching_left

    #1 is the only value for which an enclosed mismatch is ambiguous:
    #0   -> bulge has no enclosed mismatch
    #>=2 -> . can only go with one | since 1 == nb_enclosed

    contribution = mismatch_contribution(base_next_left_prev,base_next_left,
    base_right_matching_next_left_next)

    circuit.append(substract_cst_gate(contribution,nb_bits_energy).
                   control(
                       1+3*nb_bits_base,ctrl_state=
                       base_to_ctrl_state(base_next_left_prev)+
                       base_to_ctrl_state(base_next_left)+
                       base_to_ctrl_state(base_right_matching_next_left_next)+
                       "0"
                   ),
                   [should_not_cancel]+
                   [next_left_prev[i] for i in range(nb_bits_base)]+
                   [next_left[i] for i in range(nb_bits_base)]+
                   [right_matching_next_left_next[i] for i in range(nb_bits_base)]+

                   [energy[i] for i in range(nb_bits_energy)]
                  )
    return "cancel enclosed mismatch"


def add_outside_mismatch_contribution(circuit,
                                      folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
    incorrect_configuration,energy,
                                      all_registers,
                                      nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                                      all_nb_bits,
                                      incr_position,decr_position,add_position,
    incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                                      all_gates,
                                      length,sequence_specification,folding_specification,
    loop_specification,
                                      all_specifiers,
                                      basic_args_to_call_append,
                                      k_left,
                                      base_left_prev,base_left,base_right_matching_left_next
    ):
    #parameters base_ are in the order of the sequence

    will_use_registers(circuit,[folding,sequence,continue_searching_right_matching_left,
    right_matching_left_next,use_right_matching_left_next,energy])

    is_not_outside_mismatch = continue_searching_right_matching_left
    contribution = mismatch_contribution(base_left_prev,base_left,
    base_right_matching_left_next)
```

```
87
88      circuit.append(add_cst_gate(contribution,nb_bits_energy).
89                     control(
90                         1+3*nb_bits_base,ctrl_state=
91                         base_to_ctrl_state(base_left_prev)+
92                         base_to_ctrl_state(base_left)+
93                         base_to_ctrl_state(base_right_matching_left_next)+
94                         "0"
95                     ),
96                     [is_not_outside_mismatch]+
97                     [sequence[k_left-1][i] for i in range(nb_bits_base)]+
98                     [sequence[k_left  ][i] for i in range(nb_bits_base)]+
99                     [right_matching_left_next[i] for i in range(nb_bits_base)]+
100                    [energy[i] for i in range(nb_bits_energy)]
101                   )
102     return "outside mismatch"
103
104
105 def add_enclosing_mismatch_contribution(circuit,
106                                         folding,sequence,count,
        continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
        next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
        right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
        incorrect_configuration,energy,
107                                         all_registers,
108                                         nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
109                                         all_nb_bits,
110                                         incr_position,decr_position,add_position,
        incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
111                                         all_gates,
112                                         length,sequence_specification,folding_specification,
        loop_specification,
113                                         all_specifiers,
114                                         basic_args_to_call_append,
115                                         k_left,
116                                         base_left_next,base_right_matching_left_prev,
        base_right_matching_left):
117     #parameters base_ are in the order of the sequence
118     will_use_registers(circuit,[folding,sequence,continue_searching_right_matching_left,
        nb_enclosed,right_matching_left_prev,use_right_matching_left_prev,energy])
119
120     is_not_particular_case = continue_searching_right_matching_left
121     base_left = base_matching_Watson_Crick(base_right_matching_left)
122
123     contribution = mismatch_contribution(base_right_matching_left_prev,
        base_right_matching_left,base_left_next)
124
125     if 0 < nb_bits_enclosed:
126         #<****> what has changed is
127         #base_left_next,base_right_matching_left_prev
128         #U,U == 11,11 -> 00,11 == A,U
129         #G,U == 10,11 -> 01,11 == C,U
130         #C,U == 01,11 -> 10,11 == G,U
131         #A,U == 00,11 -> 11,11 == U,U
132
133         #U,C == 11,01 -> 00,01 == A,C
134         #G,C == 10,01 -> 01,01 == C,C
135         #C,C == 01,01 -> 10,01 == G,C
136         #A,C == 00,01 -> 11,01 == U,C
137
138         if "U" == base_left_next or "C" == base_left_next:
```

```
            if   "A" == base_right_matching_left_prev:
                base_right_matching_left_prev = "U"
            elif "C" == base_right_matching_left_prev:
                base_right_matching_left_prev = "G"
            elif "G" == base_right_matching_left_prev:
                base_right_matching_left_prev = "C"
            elif "U" == base_right_matching_left_prev:
                base_right_matching_left_prev = "A"

    circuit.append(add_cst_gate(contribution,nb_bits_energy).
                   control(
                       1+2*nb_bits_bracket_dot+1+3*nb_bits_base,ctrl_state=
                       base_to_ctrl_state(base_right_matching_left_prev)+
                       base_to_ctrl_state(base_left_next)+
                       base_to_ctrl_state(base_left)+

                       "1"+
                       bracket_dot_to_ctrl_state("<-.")+
                       bracket_dot_to_ctrl_state("(")+

                       "1"
                   ),
                   [is_not_particular_case]+
                   [folding[k_left  ][i] for i in range(nb_bits_bracket_dot)]+
                   [folding[k_left+1][i] for i in range(nb_bits_bracket_dot)]+
                   [use_right_matching_left_prev]+

                   [sequence[k_left  ][i] for i in range(nb_bits_base)]+
                   [sequence[k_left+1][i] for i in range(nb_bits_base)]+
                   [right_matching_left_prev[i] for i in range(nb_bits_base)]+

                   [energy[i] for i in range(nb_bits_energy)]
                  )
    return "regular enclosing mismatch"


def stack_Watson_Crick_contribution(base_left5,base_left3):
    #the AU_end_penalty of the exterior base pair is added always (even if it is not an
    outside end)
    #and is substacted when an AU base pair is detected not being an outside end

    #so  if "A" or "U" == base_left5 then AU_end_penalty is added
    #and if "A" or "U" == base_left3 then AU_end_penalty is substracted

    return [
        #"A" == base_left5
        [-0.9               ,-2.2+AU_end_penalty,-2.1+AU_end_penalty,-1.1               ],
        #"C" == base_left
        [-2.1-AU_end_penalty,-3.3               ,-2.4               ,-2.1-AU_end_penalty],
        #"G" == base_left
        [-2.4-AU_end_penalty,-3.4               ,-3.3               ,-2.2-AU_end_penalty],
        #"U" == base_left
        [-1.3               ,-2.4+AU_end_penalty,-2.1+AU_end_penalty,-0.9               ]
    ][int(base_to_ctrl_state(base_left5),2)][int(base_to_ctrl_state(base_left3)    ,2)]


def add_stack_Watson_Crick_contribution(circuit,
                                        folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
```

```
    incorrect_configuration,energy,
                                        all_registers,
                                        nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                                        all_nb_bits,
                                        incr_position,decr_position,add_position,
    incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                                        all_gates,
                                        length,sequence_specification,folding_specification,
    loop_specification,
                                        all_specifiers,
                                        basic_args_to_call_append,
                                        k_left,
                                        base_left,base_next_left):
    will_use_registers(circuit,[folding,sequence,continue_searching_right_matching_left,
    energy])

    is_stack_or_single_bulge = continue_searching_right_matching_left

    contribution = stack_Watson_Crick_contribution(base_left,base_next_left)

    circuit.append(add_cst_gate(contribution,nb_bits_energy).
                   control(
                       1+2*nb_bits_base,ctrl_state=
                       base_to_ctrl_state(base_next_left)+
                       base_to_ctrl_state(base_left)+
                       "1"
                   ),
                   [is_stack_or_single_bulge]+
                   [sequence[k_left  ][i] for i in range(nb_bits_base)]+
                   [sequence[k_left+1][i] for i in range(nb_bits_base)]+

                   [energy[i] for i in range(nb_bits_energy)]
                  )
    return "Watson-Crick"


def add_loop_energy(circuit,
                    folding,sequence,count,continue_searching_right_matching_left,
    nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,
    right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,
    use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy
    ,
                    all_registers,
                    nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                    all_nb_bits,
                    incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,
    incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                    all_gates,
                    length,sequence_specification,folding_specification,loop_specification,
                    all_specifiers,
                    basic_args_to_call_append,
                    k_left):
    will_use_registers(circuit,[folding,sequence,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
    energy])

    #use https://rna.urmc.rochester.edu/NNDB/tutorials.html
    #sequence[0] is 5' --> sequence[length-1] is 3'

    bases = ["A","C","G","U"]
```

```
245
246
247     # 1) loop_length
248
249     #TODO
250     #contributions of:
251     # - loop_length
252     # - asymmetry in interior loop
253     # - missing AU_end_penalty on the exterior side of helixes
254     # - AU_end_penalty is different in interior loops -> add the difference
255     # - dandling end
256
257
258     # 2) Hairpin
259
260     is_hairpin_mismatch = continue_searching_right_matching_left
261
262     #hairpin loop with 3 == loop_lengh do not have mismatch
263
264     # 2.1) UU, GA ou AG interior mismatch
265     # 2.1.1) Detect
266
267     #U,U == 11,11 -> 11,11
268     #G,A == 10,00 -> 11,01
269     #A,G == 00,10 -> 01,11
270     circuit.cx(sequence[k_left+1][1],sequence[k_left+1][0])
271     circuit.cx(sequence[k_left+1][1],right_matching_left_prev[0])
272     circuit.cx(right_matching_left_prev[1],sequence[k_left+1][0])
273     circuit.cx(right_matching_left_prev[1],right_matching_left_prev[0])
274
275     circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2             ,ctrl_state="
    11"  +n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[
    k_left][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[sequence[k_left+1][0],right_matching_left_prev[0],
    is_hairpin_mismatch])
276     circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base,ctrl_state="
    0101"+n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[
    k_left][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[sequence[k_left+1][i] for i in range(nb_bits_base)]+[
    right_matching_left_prev[i] for i in range(nb_bits_base)]+[is_hairpin_mismatch])
277
278     #but hairpin loop with 3 >= loop_lengh do not have mismatch
279     circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base+(
    nb_bits_position-2),ctrl_state=n_to_ctrl_state(0,nb_bits_position-2)+"0101"+
    n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[k_left][i]
    for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i
    in range(nb_bits_base)]+[loop_length[i] for i in range(2,nb_bits_position)]+[
    is_hairpin_mismatch])
280
281
282     # 2.1.2) Apply contribution
283
284     circuit.append(add_cst_gate(hairpin_mismatch_UU_AG_contribution,nb_bits_energy).control
    (),[is_hairpin_mismatch]+[energy[i] for i in range(nb_bits_energy)])
285
286
287     # 2.1.3) Restore
288
289     circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base+(
    nb_bits_position-2),ctrl_state=n_to_ctrl_state(0,nb_bits_position-2)+"0101"+
```

```
    n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[k_left][i]
    for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i
    in range(nb_bits_base)]+[loop_length[i] for i in range(2,nb_bits_position)]+[
    is_hairpin_mismatch])
290 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base,ctrl_state="
    0101"+n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[
    k_left][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[sequence[k_left+1][i] for i in range(nb_bits_base)]+[
    right_matching_left_prev[i] for i in range(nb_bits_base)]+[is_hairpin_mismatch])
291 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2             ,ctrl_state="
    11"  +n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("(")),[folding[
    k_left][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[sequence[k_left+1][0],right_matching_left_prev[0],
    is_hairpin_mismatch])
292 circuit.cx(right_matching_left_prev[1],right_matching_left_prev[0])
293 circuit.cx(right_matching_left_prev[1],sequence[k_left+1][0])
294 circuit.cx(sequence[k_left+1][1],right_matching_left_prev[0])
295 circuit.cx(sequence[k_left+1][1],sequence[k_left+1][0])
296
297
298 # 2.2) GG interior mismatch
299
300 #detect
301 is_hairpin_mismatch_GG = continue_searching_right_matching_left
302 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base,ctrl_state=
    base_to_ctrl_state("G")+base_to_ctrl_state("G")+n_to_ctrl_state(0,nb_bits_enclosed)+
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[sequence[k_left+1][i] for i in
    range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    is_hairpin_mismatch])
303
304 #but hairpin loop with 3 >= loop_lengh do not have mismatch
305 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base+(
    nb_bits_position-2),ctrl_state=n_to_ctrl_state(0,nb_bits_position-2)+base_to_ctrl_state(
    "G")+base_to_ctrl_state("G")+n_to_ctrl_state(0,nb_bits_enclosed)+
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[sequence[k_left+1][i] for i in
    range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    loop_length[i] for i in range(2,nb_bits_position)]+[is_hairpin_mismatch])
306
307 #apply contribution
308 circuit.append(add_cst_gate(hairpin_mismatch_GG_contribution,nb_bits_energy).control(),[
    is_hairpin_mismatch]+[energy[i] for i in range(nb_bits_energy)])
309
310 #restore
311 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base+(
    nb_bits_position-2),ctrl_state=n_to_ctrl_state(0,nb_bits_position-2)+base_to_ctrl_state(
    "G")+base_to_ctrl_state("G")+n_to_ctrl_state(0,nb_bits_enclosed)+
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[sequence[k_left+1][i] for i in
    range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    loop_length[i] for i in range(2,nb_bits_position)]+[is_hairpin_mismatch])
312 circuit.append(MCXGate(nb_bits_bracket_dot+nb_bits_enclosed+2*nb_bits_base,ctrl_state=
    base_to_ctrl_state("G")+base_to_ctrl_state("G")+n_to_ctrl_state(0,nb_bits_enclosed)+
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[sequence[k_left+1][i] for i in
    range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    is_hairpin_mismatch])
313
314
```

```
    # 2.3) C3 hairpin loop

    circuit.append(add_cst_gate(hairpin_C3_contribution,nb_bits_energy).control(
nb_bits_bracket_dot+nb_bits_enclosed+(nb_bits_position-2)+3*nb_bits_base,ctrl_state=
base_to_ctrl_state("C")+base_to_ctrl_state("C")+base_to_ctrl_state("C")+n_to_ctrl_state
(0,nb_bits_position-2)+n_to_ctrl_state(0,nb_bits_enclosed)+bracket_dot_to_ctrl_state("("
)),[folding[k_left][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in
range(nb_bits_enclosed)]+[loop_length[i] for i in range(2,nb_bits_position)]+[sequence[
k_left+1][i] for i in range(nb_bits_base)]+[sequence[k_left+2][i] for i in range(
nb_bits_base)]+[sequence[k_left+3][i] for i in range(nb_bits_base)]+[energy[i] for i in
range(nb_bits_energy)])


    # 2.4) all-C hairpin loop with 3 < loop_length (ignored)



    if 0 < nb_bits_enclosed:
        # 3) Multibranch loop

        #use https://rna.urmc.rochester.edu/NNDB/turner04/mb.html


        if 1 < nb_bits_enclosed:
            # 3.1) Constant

            #<*****> energy has been initialised to (length-1-min_length_hairpin_loop)*
multiloop_flat_contribution
            #so multiloop_flat_contribution should be substracted if there is no multiloop
starting at position k_left
            circuit.append(substract_cst_gate(multiloop_flat_contribution,nb_bits_energy).
control(nb_bits_enclosed-1,ctrl_state=n_to_ctrl_state(0,nb_bits_enclosed-1)),[
nb_enclosed[i] for i in range(1,nb_bits_enclosed)]+[energy[i] for i in range(
nb_bits_energy)])


        # 3.2) Average asmmetry (ignored)


        # 3.3) Number of branches

        #add nb_enclosed*one_branch_contribution to energy
        for i in range(nb_bits_enclosed):
            for _ in range(2**i):
                circuit.append(add_cst_gate(one_branch_contribution,nb_bits_energy).control
(),[nb_enclosed[i]]+[energy[i] for i in range(nb_bits_energy)])

        #a loop that is not a multiloop may have 1 enclosed branch, so this contribution
should be cancelled
        circuit.append(substract_cst_gate(one_branch_contribution,nb_bits_energy).control(
nb_bits_enclosed,ctrl_state=n_to_ctrl_state(1,nb_bits_enclosed)),[nb_enclosed[i] for i
in range(nb_bits_enclosed)]+[energy[i] for i in range(nb_bits_energy)])


        # 3.4) Particular case of strain (three-way branching loops with fewer than two
unpaired nucleotides)

        #<***> control by 2 > loop_length
        circuit.append(add_cst_gate(strain_contribution,nb_bits_energy).control(
nb_bits_enclosed+(nb_bits_position-1),ctrl_state=n_to_ctrl_state(0,nb_bits_position-1)+
n_to_ctrl_state(2,nb_bits_enclosed)),[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[
```

```
    loop_length[i] for i in range(1,nb_bits_position)]+[energy[i] for i in range(
    nb_bits_energy)])
356
357
358         # 4) Stack and single nucleotide bulge loop
359
360         #use https://rna.urmc.rochester.edu/NNDB/turner04/wc.html
361         #and https://rna.urmc.rochester.edu/NNDB/turner04/bulge.html
362         #this RNA is not a duplex, so there is no symmetric duplex term
363         #(if this is a misunderstanding, then it is actually a simplification choice!)
364         #the absence of AU_end_penalty for single nucleotide bulge loop has not been
    forgotten
365
366
367         # 4.1) Stack
368
369         #use continue_searching_right_matching_left as an auxillary qbit
370         #(that should be cleaned after use)
371         is_stack_or_single_bulge = continue_searching_right_matching_left
372         circuit.append(MCXGate(nb_bits_enclosed+(nb_bits_position-1),ctrl_state=
    n_to_ctrl_state(0,nb_bits_position-1)+n_to_ctrl_state(1,nb_bits_enclosed)),[nb_enclosed[
    i] for i in range(nb_bits_enclosed)]+[loop_length[i] for i in range(1,nb_bits_position)
    ]+[is_stack_or_single_bulge])
373         for base_left in bases:
374             for base_next_left in bases:
375                 add_stack_Watson_Crick_contribution(*basic_args_to_call_append,k_left,
    base_left,base_next_left)
376
377         #restore is_stack_or_single_bulge
378         circuit.append(MCXGate(nb_bits_enclosed+(nb_bits_position-1),ctrl_state=
    n_to_ctrl_state(0,nb_bits_position-1)+n_to_ctrl_state(1,nb_bits_enclosed)),[nb_enclosed[
    i] for i in range(nb_bits_enclosed)]+[loop_length[i] for i in range(1,nb_bits_position)
    ]+[is_stack_or_single_bulge])
379
380
381         # 4.2) Special C bulge
382         # 4.2.1) Detect
383
384         #a special C bulge is a "bulged C [residue] adjacent to at least one C" (from https
    ://www.pnas.org/doi/10.1073/pnas.0401799101)
385
386         is_special_C = continue_searching_right_matching_left
387
388         #C == sequence[k_left]
389         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("C")+bracket_dot_to_ctrl_state(".")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
    k_left  ][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
390         #C == sequence[k_left+2]
391         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("C")+bracket_dot_to_ctrl_state(".")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
    k_left+2][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
392         #correct C == sequence[k_left] and C == sequence[k_left+2] redundancy
393         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+3*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("C")+base_to_ctrl_state("C")+
    bracket_dot_to_ctrl_state(".")+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,
```

```
    nb_bits_enclosed)),[sequence[k_left][i] for i in range(nb_bits_base)]+[sequence[k_left
    +2][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(nb_bits_base)
    ]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)]+[nb_enclosed[
    i] for i in range(nb_bits_enclosed)]+[is_special_C])
394
395         #G == sequence[k_left] (so C == right_matching_left)
396         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+bracket_dot_to_ctrl_state("|")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
    k_left][i] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
397         #G == next_left (so C == right_matching_next_left)
398         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+bracket_dot_to_ctrl_state("|")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[next_left[i
        ] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
399         #correct #G == sequence[k_left] and G == next_left redundancy
400         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+3*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+base_to_ctrl_state("G")+
    bracket_dot_to_ctrl_state("|")+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,
    nb_bits_enclosed)),[sequence[k_left][i] for i in range(nb_bits_base)]+[next_left[i] for
    i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)]+[nb_enclosed[i]
    for i in range(nb_bits_enclosed)]+[is_special_C])
401
402
403         # 4.2.2) Apply contribution
404
405         circuit.append(add_cst_gate(special_C_bulge_bonus,nb_bits_energy).control(),[
    is_special_C]+[energy[i] for i in range(nb_bits_energy)])
406
407
408         # 4.2.3) Restore
409
410         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+3*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+base_to_ctrl_state("G")+
    bracket_dot_to_ctrl_state("|")+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,
    nb_bits_enclosed)),[sequence[k_left][i] for i in range(nb_bits_base)]+[next_left[i] for
    i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(nb_bits_base)]+[
    folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)]+[nb_enclosed[i]
    for i in range(nb_bits_enclosed)]+[is_special_C])
411         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+bracket_dot_to_ctrl_state("|")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[next_left[i
        ] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
412         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("G")+bracket_dot_to_ctrl_state("|")+
    n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
    k_left][i] for i in range(nb_bits_base)]+[right_matching_left_prev[i] for i in range(
    nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
    ]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
413         circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+3*nb_bits_base,ctrl_state
    =base_to_ctrl_state("C")+base_to_ctrl_state("C")+base_to_ctrl_state("C")+
    bracket_dot_to_ctrl_state(".")+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,
    nb_bits_enclosed)),[sequence[k_left][i] for i in range(nb_bits_base)]+[sequence[k_left
    +2][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(nb_bits_base)
```

```
]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)]+[nb_enclosed[
i] for i in range(nb_bits_enclosed)]+[is_special_C])
    circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
=base_to_ctrl_state("C")+base_to_ctrl_state("C")+bracket_dot_to_ctrl_state(".")+
n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
k_left+2][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(
nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])
    circuit.append(MCXGate(nb_bits_enclosed+nb_bits_position+1+2*nb_bits_base,ctrl_state
=base_to_ctrl_state("C")+base_to_ctrl_state("C")+bracket_dot_to_ctrl_state(".")+
n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(1,nb_bits_enclosed)),[sequence[
k_left  ][i] for i in range(nb_bits_base)]+[sequence[k_left+1][i] for i in range(
nb_bits_base)]+[folding[k_left+1][0]]+[loop_length[i] for i in range(nb_bits_position)
]+[nb_enclosed[i] for i in range(nb_bits_enclosed)]+[is_special_C])


    # 4.3) Degeneracy of bulge (ignored)


# 5) Flat contribution per helix

#for helix_flat_contribution = 4.09
#use the intermolecular initiation term from https://rna.urmc.rochester.edu/NNDB/
turner04/wc.html


# 5.1) Detect an end of helix

#use continue_searching_right_matching_left as an auxillary qbit
#(that should be cleaned after use)
is_helix_interior_end = continue_searching_right_matching_left

#set is_helix_interior_end = 2 > nb_enclosed and 0 == loop_length
if 1 < nb_bits_enclosed:
    circuit.append(MCXGate((nb_bits_enclosed-1)+nb_bits_position,ctrl_state=
n_to_ctrl_state(0,nb_bits_position)+n_to_ctrl_state(0,nb_bits_enclosed-1)),[nb_enclosed[
i] for i in range(1,nb_bits_enclosed)]+[loop_length[i] for i in range(nb_bits_position)
]+[is_helix_interior_end])
else:
    #2 > nb_enclosed is true
    circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(0,
nb_bits_position)),[loop_length[i] for i in range(nb_bits_position)]+[
is_helix_interior_end])

#and flip the result to get is_helix_interior_end = 2 <= nb_enclosed or 0 < loop_length
circuit.x(is_helix_interior_end)


# 5.2) Add the contribution

circuit.append(add_cst_gate(helix_flat_contribution,nb_bits_energy).control(),[
is_helix_interior_end]+[energy[i] for i in range(nb_bits_energy)])


# 5.3) Restore is_helix_interior_end

if 1 < nb_bits_enclosed:
    circuit.append(MCXGate((nb_bits_enclosed-1)+nb_bits_position,ctrl_state=
n_to_ctrl_state(0,nb_bits_position)+n_to_ctrl_state(0,nb_bits_enclosed-1)),[nb_enclosed[
i] for i in range(1,nb_bits_enclosed)]+[loop_length[i] for i in range(nb_bits_position)
]+[is_helix_interior_end])
```

```
    else:
        circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(0,
    nb_bits_position)),[loop_length[i] for i in range(nb_bits_position)]+[
    is_helix_interior_end])

    #continue_searching_right_matching_left has been restored
    #except that now 1 == continue_searching_right_matching_left


    if 0 < nb_bits_enclosed:
        # 6) substract loop_length_right to loop_length

        #use continue_searching_right_matching_left as an auxillary qbit
        #(that should be cleaned after use)
        #initially 1 == continue_searching_right_matching_left
        initial_carry=continue_searching_right_matching_left
        circuit.x(loop_length_right)
        circuit.append(add_position,[initial_carry]+[loop_length_right[i] for i in range(
    nb_bits_position)]+[loop_length[i] for i in range(nb_bits_position)])

        #now loop_length is used as loop_length_left
        loop_length_left = loop_length



        # 7) Other tow-branched loop (bulge and interior loop)

        #these are loops with 1 == nb_enclosed and 0 < loop_length

        should_not_cancel_mismatch = continue_searching_right_matching_left


        # 7.1) Particular cases 1*1 interior loop
        # 7.1.1) Cancel potential contribution that will come from 8.1) with a greater
    k_left

        #detect
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed+(nb_bits_bracket_dot-1)
    +1,ctrl_state="01"+n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,
    nb_bits_position)+n_to_ctrl_state(1,nb_bits_position)),[loop_length_left[i] for i in
    range(nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[
    nb_enclosed[i] for i in range(nb_bits_enclosed)]+[folding[k_left+1][nb_bits_bracket_dot
    -1],use_right_matching_left_prev,should_not_cancel_mismatch])

        for base_next_left_prev in bases:
            for base_next_left in bases:
                for base_right_matching_next_left_next in bases:
                    substract_enclosed_mismatch_contribution(*basic_args_to_call_append,
    k_left,base_next_left_prev,base_next_left,base_right_matching_next_left_next)

        #restore
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed+(nb_bits_bracket_dot-1)
    +1,ctrl_state="01"+n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,
    nb_bits_position)+n_to_ctrl_state(1,nb_bits_position)),[loop_length_left[i] for i in
    range(nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[
    nb_enclosed[i] for i in range(nb_bits_enclosed)]+[folding[k_left+1][nb_bits_bracket_dot
    -1],use_right_matching_left_prev,should_not_cancel_mismatch])

        # 7.1.2) Contribution of these particular cases

        is_not_11 = continue_searching_right_matching_left
```

```
499
500         circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_11])
501
502         for base_left in bases:
503             for base_interior_left in bases:
504                 for base_next_left in bases:
505                     for base_interior_right in bases:
506                         #TODO: easy but boring
507                         #use https://rna.urmc.rochester.edu/NNDB/turner04/int11.txt
508                         pass
509
510         #restore
511         circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_11])
512
513
514         # 7.2) Particular cases 1*2 and 2*1 interior loop
515         # 7.2.1) Cancel potential contribution that will come from 8.1) with a greater
    k_left
516
517         #detect
518
519         #make 1*2 look like 2*1
520         #when 2*1
521         #   <-. == folding[k_left+1]
522         #   0 == folding[k_left+1][1]
523         #   this conserves use_right_matching_left_next
524         #when 1*2
525         #   1 == use_right_matching_left_next
526         #   this sets use_right_matching_left_next=folding[k_left+1][1]
527         #then to use use_right_matching_left_next for the **enclosed** (not the enclosing)
    mismatch,
528         #it should be used negatively controlled
529         circuit.cx(folding[k_left+1][1],use_right_matching_left_next,ctrl_state=0)
530
531         circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed+1,ctrl_state="0"+
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[use_right_matching_left_prev,should_not_cancel_mismatch])
532
533         for base_next_left_prev in bases:
534             for base_next_left in bases:
535                 for base_right_matching_next_left_next in bases:
536                     substract_enclosed_mismatch_contribution(*basic_args_to_call_append,
    k_left,base_next_left_prev,base_next_left,base_right_matching_next_left_next)
537
538
539         #restore
540         circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed+1,ctrl_state="0"+
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[use_right_matching_left_prev,should_not_cancel_mismatch])
```

```
        circuit.cx(folding[k_left+1][1],use_right_matching_left_next,ctrl_state=0)


        # 7.2.2) Contribution of 1*2 particular cases

        is_not_12 = continue_searching_right_matching_left

        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(2,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_12])

        for base_left in bases:
            for base_interior_left in bases:
                for base_next_left in bases:
                    for base_interior_right5 in bases:
                        for base_interior_right3 in bases:
                            #TODO: easy but boring and long
                            #use https://rna.urmc.rochester.edu/NNDB/turner04/int21.txt
                            pass

        #restore
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(2,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_12])


        # 7.2.3) Contribution of 2*1 particular cases

        is_not_21 = continue_searching_right_matching_left

        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(2,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_21])

        for base_left in bases:
            for base_interior_left5 in bases:
                for base_interior_left3 in bases:
                    for base_next_left in bases:
                        for base_interior_right in bases:
                            #TODO: easy but boring and long
                            #use https://rna.urmc.rochester.edu/NNDB/turner04/int21.txt
                            #could probably be merged with 7.2.2)
                            pass

        #restore
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(2,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_21])


        # 7.3) Particular cases 2*2 interior loop
        # 7.3.1) Cancel potential contribution that will come from 8.1) with a greater
    k_left
```

```
        is_not_22 = should_not_cancel_mismatch

        #detect
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_22])

        for base_next_left_prev in bases:
            for base_next_left in bases:
                for base_right_matching_next_left_next in bases:
                    substract_enclosed_mismatch_contribution(*basic_args_to_call_append,
    k_left,base_next_left_prev,base_next_left,base_right_matching_next_left_next)


        # 7.3.2) Contribution of these particular cases

        for base_left in bases:
            for base_interior_left5 in bases:
                for base_interior_left3 in bases:
                    for base_next_left in bases:
                        for base_interior_right5 in bases:
                            for base_interior_right3 in bases:
                                #TODO: easy but boring and very long
                                #use https://rna.urmc.rochester.edu/NNDB/turner04/int22.txt
                                pass

        #restore
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_enclosed,ctrl_state=
    n_to_ctrl_state(1,nb_bits_enclosed)+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state
    (1,nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)]+[is_not_22])


    # 8) Mismatch
    # 8.1) Enclosed (particular cases compensated in 7.x.1))

    #let's consider the mismatch outside of the loop enclosed by ( == folding[k_left]
    #add its contribution whatever the case
    #and if it happends it was a particular case of enclosed mismatch
    #then it has been corrected when this loop was treated in 3) at a different k_left

    #folding[k_left] == ( followed by ...)<-. is the shortest way to get a mismatch outside
    of a loop
    if 0 < k_left and k_left < length-2-min_length_hairpin_loop:

        is_not_outside_mismatch = continue_searching_right_matching_left

        circuit.append(MCXGate(2*nb_bits_bracket_dot+1,ctrl_state="1"+
    bracket_dot_to_ctrl_state("(")+bracket_dot_to_ctrl_state(".->")),[folding[k_left-1][i]
    for i in range(nb_bits_bracket_dot)]+[folding[k_left  ][i] for i in range(
    nb_bits_bracket_dot)]+[use_right_matching_left_next,is_not_outside_mismatch])

        for base_left_prev in bases:
            for base_left in bases:
                for base_right_matching_left_next in bases:
                    add_outside_mismatch_contribution(*basic_args_to_call_append,k_left,
    base_left_prev,base_left,base_right_matching_left_next)
```

```
    # 8.2) Enclosing (no particular cases)

    #there is no particular case if there is only the exterior loop and maybe an hairpin
    loop
    if 0 < nb_bits_enclosed:
        # 8.2.1) Detect non particular cases

        #use continue_searching_right_matching_left as an auxillary qbit
        #(that should be cleaned after use)
        #initially 1 == continue_searching_right_matching_left
        is_not_particular_case = continue_searching_right_matching_left

        #1*_ interior loop are particular cases
        #there is no mismatch in bulges (so do not care about _*0)
        circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    is_not_particular_case])

        #_*1 interior loop are particular cases
        #there is no mismatch in bulges (so do not care about 0*_)
        circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_right[i] for i in range(nb_bits_position)]+[
    is_not_particular_case])

        #before correcting for redundancy with 1*1
        #make 2*2 and 1*1 look like 1*1 over one less qbit like this
        #0 -> 0
        #1 -> 3
        #2 -> 2
        #3 -> 1
        circuit.cx(loop_length_left[0],loop_length_left[1])
        circuit.cx(loop_length_right[0],loop_length_right[1])

        #correct redundancy with 1*1
        #and handle 2*2 that are also particular cases
        circuit.append(MCXGate(2*(nb_bits_position-1),ctrl_state=n_to_ctrl_state(1,
    nb_bits_position-1)+n_to_ctrl_state(1,nb_bits_position-1)),[loop_length_left[i] for i in
     range(1,nb_bits_position)]+[loop_length_right[i] for i in range(1,nb_bits_position)]+[
    is_not_particular_case])

        #2*3 with GA or GG mismatch are particualr cases
        #but 2*3 -> 2*1 now
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="0"+base_to_ctrl_state("G")+n_to_ctrl_state(1,nb_bits_position)+
    n_to_ctrl_state(2,nb_bits_position)),[loop_length_left[i] for i in range(
    nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[sequence[
    k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[0],
    is_not_particular_case])

        #3*2 with GA or GG mismatch are particular cases
        #but 3*2 -> 1*2 now
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="0"+base_to_ctrl_state("G")+n_to_ctrl_state(2,nb_bits_position)+
    n_to_ctrl_state(1,nb_bits_position)),[loop_length_left[i] for i in range(
    nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[sequence[
    k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[0],
    is_not_particular_case])

        #A,G == 00,10 -> 00,10
```

```
        #U,U == 11,11 -> 00,11
        #in order to control by 00,1x
        for i in range(nb_bits_bracket_dot):
            circuit.cx(right_matching_left_prev[0],sequence[k_left+1][i])

        #so what has changed is
        #U,U == 11,11 -> 00,11 == A,U
        #G,U == 10,11 -> 01,11 == C,U
        #C,U == 01,11 -> 10,11 == G,U
        #A,U == 00,11 -> 11,11 == U,U

        #U,C == 11,01 -> 00,01 == A,C
        #G,C == 10,01 -> 01,01 == C,C
        #C,C == 01,01 -> 10,01 == G,C
        #A,C == 00,01 -> 11,01 == U,C

        #it will be used there <****>

        #2*3 with UU or AG mismatch are particular cases
        #but 2*3 -> 2*1 now
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="001"+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(2,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[right_matching_left_prev[0]]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[is_not_particular_case])

        #3*2 with UU or AG mismatch are particular cases
        #but 3*2 -> 1*2 now
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="001"+n_to_ctrl_state(2,nb_bits_position)+n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[right_matching_left_prev[0]]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[is_not_particular_case])


    # 8.2.2) Handle non particular cases

    for base_left_next in bases:
        for base_right_matching_left_prev in bases:
            for base_right_matching_left in bases:
                add_enclosing_mismatch_contribution(*basic_args_to_call_append,k_left,
    base_left_next,base_right_matching_left_prev,base_right_matching_left)


    if 0 < nb_bits_enclosed:
        # 8.2.3) Restore everything

        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="001"+n_to_ctrl_state(2,nb_bits_position)+n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[right_matching_left_prev[0]]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[is_not_particular_case])
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="001"+n_to_ctrl_state(1,nb_bits_position)+n_to_ctrl_state(2,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    loop_length_right[i] for i in range(nb_bits_position)]+[right_matching_left_prev[0]]+[
    sequence[k_left+1][i] for i in range(nb_bits_base)]+[is_not_particular_case])

        #OPTIMISATION: restoring this could be avoided by maintaining a global variable
    about permutation of base encoding
        #00,10 -> 00,10
```

```
        #11,11 -> 00,11
        for i in range(nb_bits_bracket_dot):
            circuit.cx(right_matching_left_prev[0],sequence[k_left+1][i])

        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="0"+base_to_ctrl_state("G")+n_to_ctrl_state(2,nb_bits_position)+
    n_to_ctrl_state(1,nb_bits_position)),[loop_length_left[i] for i in range(
    nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[sequence[
    k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[0],
    is_not_particular_case])
        circuit.append(MCXGate(2*nb_bits_position+nb_bits_bracket_dot+(nb_bits_bracket_dot
    -1),ctrl_state="0"+base_to_ctrl_state("G")+n_to_ctrl_state(1,nb_bits_position)+
    n_to_ctrl_state(2,nb_bits_position)),[loop_length_left[i] for i in range(
    nb_bits_position)]+[loop_length_right[i] for i in range(nb_bits_position)]+[sequence[
    k_left+1][i] for i in range(nb_bits_base)]+[right_matching_left_prev[0],
    is_not_particular_case])
        circuit.append(MCXGate(2*(nb_bits_position-1),ctrl_state=n_to_ctrl_state(1,
    nb_bits_position-1)+n_to_ctrl_state(1,nb_bits_position-1)),[loop_length_left[i] for i in
     range(1,nb_bits_position)]+[loop_length_right[i] for i in range(1,nb_bits_position)]+[
    is_not_particular_case])
        circuit.cx(loop_length_right[0],loop_length_right[1])
        circuit.cx(loop_length_left[0],loop_length_left[1])
        circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_right[i] for i in range(nb_bits_position)]+[
    is_not_particular_case])
        circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(1,
    nb_bits_position)),[loop_length_left[i] for i in range(nb_bits_position)]+[
    is_not_particular_case])


    circuit.x(continue_searching_right_matching_left)

    return "loop energy"
```

Listing 9 – Computation of energy. Mainly loop contribution based on length and dandling end contribution remain to be done.

```
def automaton_step(circuit,
                   folding,sequence,count,continue_searching_right_matching_left,nb_enclosed
    ,loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,
    right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,
    use_right_matching_left_next,incorrect_configuration,energy,
                   all_registers,
                   nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                   all_nb_bits,
                   incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,
    incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                   all_gates,
                   length,sequence_specification,folding_specification,loop_specification,
                   all_specifiers,
                   basic_args_to_call_append,
                   k_left,k,
                   compute_energy=True,
                   uncompute=False):
    if compute_energy:
        will_use_registers(circuit,[folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next])
        if not uncompute:
            will_use_registers(circuit,[incorrect_configuration])
    else:
```

```
        will_use_registers(circuit,[folding,count,continue_searching_right_matching_left])

    #<*> if it has been checked that there is a matching ) to find,
    #then continue_searching_right_matching_left == 1
    #otherwise continue_searching_right_matching_left == 0


    #1), 2) and 3) before 4) Update count
    #and
    #5) and 6) after 4) Update count
    #so that things are always controled by 1 == count
    #(which could be used by the compiler for optimisation when implementing sevral time in a row the same MCXGate())

    if compute_energy:


        # 1) Get next_left_prev and next_left

        for i in range(nb_bits_base):
            #if 0 < k (which is always the case when compute_energy)
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state("(")+"1"),[continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k-1][i],next_left_prev[i]])
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state("(")+"1"),[continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k  ][i],next_left     [i]])


        # 2) Get right_matching_left_prev and right_matching_left_next

        #get right_matching_left_prev and use_right_matching_left_prev
        for i in range(nb_bits_base):
            circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
         ,ctrl_state="1"                                +n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k-1][i],right_matching_left_prev[i]])
        circuit.append(        MCXGate(1+nb_bits_bracket_dot+nb_bits_position+nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state(".->")+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)]+[folding[k][i] for i in range(nb_bits_bracket_dot)]+[folding[ k-1][i] for i in range(nb_bits_bracket_dot)]+[use_right_matching_left_prev])

        #FUTURE: if one adds GU pairs, then one base of a valid base pair would not be enough anymore to know which base pair it is
        #so there would be a need for right_matching_left, this way:
        #for i in range(nb_bits_base):
        #    circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
         ,ctrl_state="1"                                 +n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k  ][i],right_matching_left[i]     ])

        #get right_matching_left_next and use_right_matching_left_next
        if length-1 > k:
            for i in range(nb_bits_base):
                circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1
```

```
     ,ctrl_state="1"                                  +n_to_ctrl_state(1,nb_bits_position)+
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[j]
for j in range(nb_bits_position)]+[folding[k][j] for j in range(nb_bits_bracket_dot)]+[
sequence[k+1][i],right_matching_left_next[i]])
        circuit.append(    MCXGate(1+nb_bits_bracket_dot+nb_bits_position+
nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state("<-.")+n_to_ctrl_state(1,
nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)]+[
folding[k][i] for i in range(nb_bits_bracket_dot)]+[folding[ k+1][i] for i in range(
nb_bits_bracket_dot)]+[use_right_matching_left_next])
    #else use_right_matching_left_next=0


    # 3) Check base pairing rules

    #FUTURE: if one adds GU pairs, then there would be a right_matching_left
    #so base pairing rules could be checked more efficiently in characterise_loop()

    #the base encoding is such that cx(sequence[k_left],sequence[k_right]) leads to
    #"11" == sequence[k_right] if and only if (sequence[k_left],sequence[k_right]) is a
valid Watson-Crick base pair
    for i in range(nb_bits_base):
        circuit.cx(sequence[k_left][i],sequence[k][i])
    if not uncompute:
        circuit.append(incr_incorrect.control(1+nb_bits_bracket_dot+nb_bits_position+
nb_bits_base,ctrl_state="11"+n_to_ctrl_state(1,nb_bits_position)+
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[count[i]
for i in range(nb_bits_position)]+[folding[k][i] for i in range(nb_bits_bracket_dot)]+[
sequence[k][i] for i in range(nb_bits_base)]+[incorrect_configuration[i] for i in range(
nb_bits_incorrect)])
    #there is actually no need to restore sequence[k]


# 4) Update count

#if 1 == continue_searching_right_matching_left
#then update count for the next candidate
#OPTIMISATION: use smaller increment and decrement circuits when the maximum value of
count enables it (since it never goes negative)
if 0 == k:
    #then -1 == k_left and ( == folding[k=-1]
    # - so 1 == continue_searching_right_matching_left
    #   so it is ok not controlling by continue_searching_right_matching_left here
    # - so 1 == count
    #   so there is no need to incr_position or decr_position in order to increment or
decrement
    circuit.append(MCXGate((nb_bits_bracket_dot-1),ctrl_state=bracket_dot_to_ctrl_state(
"|")),[folding[k][0]]+[count[0]])
    circuit.append(MCXGate( nb_bits_bracket_dot   ,ctrl_state=bracket_dot_to_ctrl_state(
"(")),[folding[k][i] for i in range(nb_bits_bracket_dot)]+[count[1]])
else:
    circuit.append(incr_position.control(1+nb_bits_bracket_dot,ctrl_state=
bracket_dot_to_ctrl_state("(")+"1"),[continue_searching_right_matching_left]+[folding[k
][i] for i in range(nb_bits_bracket_dot)]+[count[i] for i in range(nb_bits_position)])
    circuit.append(decr_position.control(1+nb_bits_bracket_dot,ctrl_state=
bracket_dot_to_ctrl_state(")")+"1"),[continue_searching_right_matching_left]+[folding[k
][i] for i in range(nb_bits_bracket_dot)]+[count[i] for i in range(nb_bits_position)])

#if 0 == continue_searching_right_matching_left then simply increment count so that it
never equals 0 again
circuit.append(incr_position.control(ctrl_state="0"),[
```

```
    continue_searching_right_matching_left]+[count[i] for i in range(nb_bits_position)])
96
97
98  if compute_energy:
99
100
101         # 5) Get right_matching_next_left_next
102
103         if k < length-1:
104             for i in range(nb_bits_base):
105                 #FUTURE: if one adds GU pairs, then one base of a valid base pair is not
    enough anymore to know which base pair it is
106                 #so there would be a need for right_matching_next_left, this way:
107                 #circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,"1"+
    n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k  ][i],
    right_matching_next_left[i]     ])
108                 circuit.append(MCXGate(1+nb_bits_bracket_dot+nb_bits_position+1,ctrl_state="
    1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][j] for j in range(nb_bits_bracket_dot)]+[sequence[k+1][i],
    right_matching_next_left_next[i]])
109
110
111         # 6) Update nb_enclosed
112
113         if 0 < nb_bits_enclosed:
114             #OPTIMISATION: use smaller increment circuit when the maximum value of
    nb_enclose enables it (which depends on min_length_hairpin_loop)
115             circuit.append(incr_nb_enclosed.control(1+nb_bits_bracket_dot+nb_bits_position,
    ctrl_state=n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(")")+"1"),[
    continue_searching_right_matching_left]+[count[j] for j in range(nb_bits_position)]+[
    folding[k][i] for i in range(nb_bits_bracket_dot)]+[nb_enclosed[i] for i in range(
    nb_bits_enclosed)])
116
117
118         # 7) Update loop_length
119
120         #loop_length is the number of . for which 1 == count
121
122         #if 1 == continue_searching_right_matching_left and 1 == count and "x0" == . ==
    folding[k]
123         #then increment loop_length
124         #OPTIMISATION: use smaller increment circuit when the maximum value of loop_length
    enables it
125         if k_left+1 == k:
126             #since the initial value 1 == loop_length is known then there is no need to
    incr_position in order to increment
127             #two CXGate() are sufficient
128             circuit.append(MCXGate(              (nb_bits_bracket_dot-1)+nb_bits_position+1,
    ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left,loop_length[0]])
129             circuit.cx(loop_length[0],loop_length[1],ctrl_state="0")
130         else:
131             circuit.append(incr_position.control((nb_bits_bracket_dot-1)+nb_bits_position+1,
    ctrl_state="1"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left]+[loop_length[i] for i in range(nb_bits_position)
    ])
```

```
132
133
134         # 8) Update loop_length_right
135
136         #in case of a non multi-loop,
137         #(that is 1 => nb_enclosed)
138         #loop_length_right is the number of . for which 1 == count after the first enclosed
    stem (if any)
139
140         #if 1 == continue_searching_right_matching_left and 1 == count and "x0" == . ==
    folding[k]
141         #and 1 == nb_enclosed (that is 1 == nb_enclosed[0] since 1 => nb_enclosed)
142         #then increment loop_length_right
143         #OPTIMISATION: use smaller increment circuit when the maximum possible value of
    loop_length enables it
144         if k_left+2+min_length_hairpin_loop < k and 0 < nb_bits_enclosed:
145             circuit.append(incr_position.control((nb_bits_bracket_dot-1)+nb_bits_position+2,
    ctrl_state="11"+n_to_ctrl_state(1,nb_bits_position)+bracket_dot_to_ctrl_state(".")),[
    folding[k][0]]+[count[i] for i in range(nb_bits_position)]+[
    continue_searching_right_matching_left,nb_enclosed[0]]+[loop_length_right[i] for i in
    range(nb_bits_position)])
146
147
148     # 9) Update continue_searching_right_matching_left
149
150     #if 0 == count
151     #then there is no more matching ) to find
152     circuit.append(MCXGate(nb_bits_position,ctrl_state=n_to_ctrl_state(0,nb_bits_position))
    ,[count[i] for i in range(nb_bits_position)]+[continue_searching_right_matching_left])
153
154     return "step"
155
156
157 def characterise_loop(circuit,
158                       folding,sequence,count,continue_searching_right_matching_left,
    nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,
    right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,
    use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy
    ,
159                       all_registers,
160                       nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
161                       all_nb_bits,
162                       incr_position,decr_position,add_position,incr_incorrect,decr_incorrect
    ,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
163                       all_gates,
164                       length,sequence_specification,folding_specification,loop_specification
    ,
165                       all_specifiers,
166                       basic_args_to_call_append,
167                       k_left,
168                       compute_energy=True,uncompute=False):
169     if compute_energy:
170         will_use_registers(circuit,[folding,sequence,count,
    continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,
    next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,
    right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,
    incorrect_configuration])
171     else:
172         will_use_registers(circuit,[folding,count,continue_searching_right_matching_left])
173
174
```

```
    # 1) Initialisation of the automaton at the position k_left

    #initialise count with folding[k_left]
    #folding[k_left] -> count
    #. == "x0"       -> 0 == "0...000"
    #do nothing
    #) == "11"       -> 0 == "0...000"
    #do nothing
    #( == "01"       -> 1 == "0...001"
    if 0 <= k_left:
        circuit.ccx(folding[k_left][0],folding[k_left][1],count[0],ctrl_state=
    bracket_dot_to_ctrl_state("("))
    else:
        circuit.x(count[0])

    #initialise continue_searching_right_matching_left=1 when 1 == count
    #indeed, when folding[k_left] == ( (which will be checked by there <*>) then there is
    initially 1 parenthesis to be closed
    circuit.cx(count[0],continue_searching_right_matching_left)


    # 2) Compute parameters by scanning the configuration with the automaton

    for k in range(k_left+1,length):
        #OPTIMISATION: break the automaton into sevral automatons to run sequencially in
    order to reduce the number of qbits
        automaton_step(*basic_args_to_call_append,k_left,k,compute_energy,uncompute)

    #Computed parameters
    #folding[k_left] -> continue_searching_right_matching_left * nb_enclosed * loop_length
    #(               -> 0 if a matching ) has been found       * nb_enclosed * number of . (
    at level 1) in the loop
    #(               -> 1 if a matching ) has not been found   * unspecified * unspecified
    #. or )          -> 0                                      * 0           * 0

    #and:
    #loop_length_right
    #next_left_prev
    #next_left
    #right_matching_next_left_next
    #right_matching_left_prev
    #right_matching_left_next

    return "characterise"
```

Listing 10 – Analysis of **`folding`**: the algorithmic core of the project. **`caracterise_folding(compute_energy=false)`** uses few enough qbits to be tested, and it works (see Section ?? and Appendix A.2).

```
def initialise_sequence(circuit,
                        folding,sequence,count,continue_searching_right_matching_left,
    nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,
    right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,
    use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy
    ,
                        all_registers,
                        nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                        all_nb_bits,
                        incr_position,decr_position,add_position,incr_incorrect,
    decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                        all_gates,
```

```
                        length,sequence_specification,folding_specification,
    loop_specification,
                        all_specifiers,
                        basic_args_to_call_append):
    will_use_registers(circuit,[sequence])

    #initialise according to a sequence_specification

    for k_left in range(length):
        if ")" != folding_specification[k_left]:
            base_specification=sequence_specification[k_left]
            #N == A or C or G or U
            if "N" == base_specification or "." == base_specification:
                circuit.h(sequence[k_left])
                pass
            elif "A" == base_specification:
                #A == "00"
                #do nothing
                pass
            #C
            elif "C" == base_specification:
                #C == "01"
                circuit.x(sequence[k_left][0])
            #G
            elif "G" == base_specification:
                #G == "10"
                circuit.x(sequence[k_left][1])
            #U
            elif "U" == base_specification:
                #U == "11"
                circuit.x(sequence[k_left])
            #R == A or G
            elif "R" == base_specification:
                circuit.h(sequence[k_left][1])
            #Y == C or U
            elif "Y" == base_specification:
                circuit.x(sequence[k_left][0])
                circuit.h(sequence[k_left][1])
            #S == G or C
            elif "S" == base_specification:
                circuit.h(sequence[k_left][0])
                circuit.cx(sequence[k_left][0],sequence[k_left][1],ctrl_state="0")
            #W == A or U
            elif "W" == base_specification:
                circuit.h(sequence[k_left][0])
                circuit.cx(sequence[k_left][0],sequence[k_left][1])
            #K == G or U
            elif "K" == base_specification:
                circuit.h(sequence[k_left][0])
                circuit.x(sequence[k_left][1])
            #M == A or C
            elif "M" == base_specification:
                circuit.h(sequence[k_left][0])
            #B == C or G or U
            else:
                change_sequence_encoding = QuantumRegister(1,name=f"change_seq_{k_left}")
                will_use_registers(circuit,[change_sequence_encoding])
                if "B" == base_specification:
                    #A == "00" -> C == "01"
                    circuit.h(sequence[k_left])
                    circuit.ccx(sequence[k_left][0],sequence[k_left][1],
```

```
    change_sequence_encoding,ctrl_state="00")
                    circuit.cx(change_sequence_encoding,sequence[k_left][0])
                    #AMPLITUDE: the amplitude of C is now twice larger than the amplitude of G or the amplitude of U
                #D == A or G or U
                elif "D" == base_specification:
                    #C == "01" -> A == "00"
                    circuit.h(sequence[k_left])
                    circuit.ccx(sequence[k_left][0],sequence[k_left][1],change_sequence_encoding,ctrl_state="01")
                    circuit.cx(change_sequence_encoding,sequence[k_left][0])
                    #AMPLITUDE: the amplitude of A is now twice larger than the amplitude of G or the amplitude of U
                #H == A or C or U
                elif "H" == base_specification:
                    #G == "10" -> C == "11"
                    circuit.h(sequence[k_left])
                    circuit.ccx(sequence[k_left][0],sequence[k_left][1],change_sequence_encoding,ctrl_state="10")
                    circuit.cx(change_sequence_encoding,sequence[k_left][0])
                    #AMPLITUDE: the amplitude of C is now twice larger than the amplitude of A or the amplitude of U
                #V == A or C or G
                elif "V" == base_specification:
                    #U == "11" -> G == "10"
                    circuit.h(sequence[k_left])
                    circuit.ccx(sequence[k_left][0],sequence[k_left][1],change_sequence_encoding,ctrl_state="11")
                    circuit.cx(change_sequence_encoding,sequence[k_left][0])
                    #AMPLITUDE: the amplitude of G is now twice larger than the amplitude of A or the amplitude of C
                else:
                    raise Exception(f"Wrong sequence specification format: \"{sequence_specification[k_left]}\" found at index {k_left} in sequence_specification.")

            #only consider sequences that would respect the base pairing rules with the folding_specification
            if "(" == folding_specification[k_left]:
                for i in range(nb_bits_base):
                    #loop_specification[k_left][0][3] is
                    #                               [3] the position
                    #                            [0] of the matching )
                    #                  [k_left] of ( == folding_specification[k_left]
                    circuit.cx(sequence[k_left][i],sequence[loop_specification[k_left][0][3]][i],ctrl_state="0")

    return "init_sequence"


def initialise_folding(circuit,
                       folding,sequence,count,continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy,
                       all_registers,
                       nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                       all_nb_bits,
                       incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                       all_gates,
```

```
                        length,sequence_specification,folding_specification,
    loop_specification,
                        all_specifiers,
                        basic_args_to_call_append):
    will_use_registers(circuit,[folding])
    #any bracket-dot symbol at any position
    for k in range(length):
        circuit.h(folding[k])

    #imagine a position -1, where folding[k_left=-1] == ( to check if the folding is well
    formed

    return "init_folding"


def RNA_design(circuit,
               folding,sequence,count,continue_searching_right_matching_left,nb_enclosed,
    loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,
    right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,
    use_right_matching_left_next,incorrect_configuration,energy,
               all_registers,
               nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
               all_nb_bits,
               incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,
    incr_nb_enclosed,incr_energy,decr_energy,add_energy,
               all_gates,
               length,sequence_specification,folding_specification,loop_specification,
               all_specifiers,
               basic_args_to_call_append):
    will_use_registers(circuit,all_registers)


    # 0) Ad hoc initialisation

    #initialise loop_length=1 because:
    # - then too small hairpin loop_length will be loop_length > 4 == 1+
    min_length_hairpin_loop so it is easy to check there <**>
    # - when doing it at the beginning, it is done in parallele whith other initialisation
    circuit.x(loop_length[0])

    #initialise energy to (length-1-min_length_hairpin_loop)*multiloop_flat_contribution
    #it is used for energy computation there <*****>
    #TODO

    # 1) Superposition at the beginning of the Grover search

    initialise_folding(*basic_args_to_call_append)
    initialise_sequence(*basic_args_to_call_append)


    # 2) Check ) matching

    #check if there is a folding[k_right] == ) non matched with a folding[k_left] == (
    #that would then match with a ( at an imaginary position folding[k_left=-1]
    #the result is in continue_searching_right_matching_left:
    # 1 == continue_searching_right_matching_left if ( == folding[k_left=-1] has not been
    matched
    # 0 == continue_searching_right_matching_left otherwise

    characterise_loop(*basic_args_to_call_append,-1,False,False)

```

```
    #increment incorrect_configuration if necessary
    #(it is necessary to increment since the value of incorrect_configuration is unknown since characterise_loop(uncompute=False) can have changed it)
    circuit.append(incr_incorrect.control(ctrl_state="0"),[continue_searching_right_matching_left]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

    inversed(characterise_loop)(*basic_args_to_call_append,-1,False,True)


    # 3) Check ( matching and add energy of loop

    for k_left in range(length-1-min_length_hairpin_loop):


        # 3.1) Compute parameters

        characterise_loop(*basic_args_to_call_append,k_left)


        # 3.2) Check ( matching

        #increment incorrect_configuration if folding[k_left] == ( has no matching )
        circuit.append(incr_incorrect.control(),[continue_searching_right_matching_left]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #TODO: set nb_bits_incorrect (with respect to the following comment)
        #sum all the incorrect flags continue_searching_right_matching_left (1 per position) into incorrect_folding
        #remark that if the flag continue_searching_right_matching_left[k_left=0] == 0 (at k_left=0 incorrect beeing 0)
        #then there is a ) somewhere
        #so continue_searching_right_matching_left[k_left=somewhere] == 0
        #so incorrect_folding is not incremented at each position
        #so the sum cannot overflow


        #continue_searching_right_matching_left has been used
        #and it can be considered that 0 == continue_searching_right_matching_left
        #(don't care about the energy if 1 == continue_searching_right_matching_left since 0 != incorrect_configuration)
        #so it can be used as an auxillary qbit
        #as far as it is cleaned after use (since inversed(characterise_loop) will be applied at the end)


        # 3.3) Check hairpin steric constraint

        #<**> too small hairpin loop_length are loop_length > 4 == 1+min_length_hairpin_loop
        #and 2 < nb_bits_position since 2+min_length_hairpin_loop == 5 <= length
        circuit.append(incr_incorrect.control((nb_bits_position-2)+nb_bits_bracket_dot,ctrl_state=bracket_dot_to_ctrl_state("(")+n_to_ctrl_state(0,nb_bits_position-2)),[loop_length[i] for i in range(2,nb_bits_position)]+[folding[k_left][i] for i in range(nb_bits_bracket_dot)]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #set loop_length to its normal value
        #in particular it makes easy to check that a loop_length is small enough for the strain_contribution to apply to certain multiloop (there <***>)
        circuit.append(decr_position,[loop_length[i] for i in range(nb_bits_position)])


```

```
        # 3.4) Compute and add energy of loop to energy

        add_loop_energy(*basic_args_to_call_append,k_left)


        # 3.4) Restore parameter qbits

        inversed(characterise_loop)(*basic_args_to_call_append,k_left)

    for k_left in range(length-1-min_length_hairpin_loop,length):
        # 3.2 again) Check ( matching
        circuit.append(incr_incorrect.control(nb_bits_bracket_dot,ctrl_state=
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

    #OPTIMISATION
    #use one less bit for folding[length-1] just by considering ./)
    #use one less bit for folding[0] just by considering ./(


    # 4) Check <-.-> choice

    #|.->. and .<-.| are prohibited
    for k_dot in range(1,length-1):
        #if 0 == folding[k_dot][0] it is a .
        #then
        # - if 1 == folding[k_dot][1] it is a .->
        #   then
        #   (a) 1 == folding[k_dot-1][0] it is a |
        #   and
        #   (b) 0 == folding[k_dot+1][0] it is a .
        #   is prohibited
        #   so by flipping (a) and (b) if 1 == folding[k_dot][1]
        #   then one get
        #   (flipped a) 0 == folding[k_dot-1][0]
        #   and
        #   (flipped b) 1 == folding[k_dot+1][0]
        #   is prohibited
        #   that is the same condition as the other case since
        #   (flipped a) == (c)
        #   and
        #   (flipped b) == (d)
        #
        # - if 0 == folding[k_dot][1] it is a <-.
        #   then
        #   (c) 0 == folding[k_dot-1][0] it is a .
        #   and
        #   (d) 1 == folding[k_dot+1][0] it is a |
        #   is prohibited

        #flip
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])

        #test
        circuit.append(incr_incorrect.control(3*(nb_bits_bracket_dot-1),ctrl_state=
    bracket_dot_to_ctrl_state("|")+bracket_dot_to_ctrl_state(".")+bracket_dot_to_ctrl_state(
    ".")),[folding[k_dot-1][0]]+[folding[k_dot][0]]+[folding[k_dot+1][0]]+[
    incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #flip back
```

```
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])


    # 5) substract the energy of the specification_folding

    #TODO: using loop_specification (which should probably be modified a bit)

    for k_left in range(length-1-min_length_hairpin_loop):
        if "(" == folding_specification[k_left]:
            k_right = loop_specification[k_left][0][3]
            loop_length_specification = loop_specification[k_left][0][3]

            #TODO


    return "RNA_design"


length = 5
folding_specification="(...)"*(length//5)+"."*(length % 5)

circ,_,_=to_circuit_builder(RNA_design)(folding_specification)

circ.draw()

#statistics of the circuit:
#print("width {} ; depth {}".format(circ.width(),circ.depth()))
```

Listing 11 – General organisation of the oracle.

A.2 Test code

```
np.set_printoptions(threshold=np.inf)

def semantics_identity(x):
    return x

def simulate_circuit(circuit,semantics=semantics_identity):
    backend = Aer.get_backend('statevector_simulator')
    nb_bits = circuit.width()
    job = execute(circuit, backend=backend, shots=1, memory=True)
    job_result = job.result()
    res=np.asarray(job_result.get_statevector(circuit))
    #state = np.asarray([(semantics(n_to_ctrl_state(i,nb_bits)),n_to_ctrl_state(i,nb_bits),
    val) for i,val in enumerate(res) if val>10**(-15)])
    state = np.asarray([(semantics(n_to_ctrl_state(i,nb_bits)),'   ') for i,val in enumerate
    (res) if val>10**(-12)])
    return circuit,state,res

def simulate(f,length):
    folding_specification="(...)"*(length//5)+"."*(length % 5)
    circ,sem,_=to_circuit_builder(f)(folding_specification)
    return simulate_circuit(circ,sem)
```

Listing 12 – Programming tool for simulations of parts of the oracle.

```
length = 6

_,state,_ = simulate(initialise_folding,length)

```

```
for (s,_) in state:
    print(s)
```

Listing 13 – **`initialise_folding()`** works.

```
length = 6

_,state,_ = simulate(initialise_sequence,length)

for (s,_) in state:
    print(s)
```

Listing 14 – **`initialise_sequence()`** works.

```
#this function is dedicated to tests
def check_folding(circuit,
                folding,sequence,count,continue_searching_right_matching_left,nb_enclosed,loop_length,loop_length_right,next_left_prev,next_left,right_matching_next_left_next,right_matching_left_prev,right_matching_left_next,use_right_matching_left_prev,use_right_matching_left_next,incorrect_configuration,energy,
                all_registers,
                nb_bits_position,nb_bits_enclosed,nb_bits_incorrect,
                all_nb_bits,
                incr_position,decr_position,add_position,incr_incorrect,decr_incorrect,incr_nb_enclosed,incr_energy,decr_energy,add_energy,
                all_gates,
                length,sequence_specification,folding_specification,loop_specification,
                all_specifiers,
                basic_args_to_call_append):

    # 1) Initiation

    initialise_folding(*basic_args_to_call_append)

    # 2) Check ) matching

    #check if there is a folding[k_right] == ) non matched with a folding[k_left] == (
    #that would then match with a ( at an imaginary position folding[k_left=-1]
    #the result is in continue_searching_right_matching_left:
    # 1 == continue_searching_right_matching_left if ( == folding[k_left=-1] has not been matched
    # 0 == continue_searching_right_matching_left otherwise

    characterise_loop(*basic_args_to_call_append,-1,False,False)

    will_use_registers(circuit,[incorrect_configuration])

    #increment incorrect_configuration if necessary
    #(it is necessary to increment since the value of incorrect_configuration is unknown since characterise_loop(uncompute=False) can have changed it)
    circuit.append(incr_incorrect.control(ctrl_state="0"),[continue_searching_right_matching_left]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])

    inversed(characterise_loop)(*basic_args_to_call_append,-1,False,True)


    # 3) Check ( matching and add energ of loop

    for k_left in range(length-1-min_length_hairpin_loop):


```

```
        # 3.1) Compute parameters

        characterise_loop(*basic_args_to_call_append,k_left,False,False)


        # 3.2) Check ( matching

        #increment incorrect_configuration if folding[k_left] == ( has no matching )
        circuit.append(incr_incorrect.control(),[continue_searching_right_matching_left]+[
    incorrect_configuration[i] for i in range(nb_bits_incorrect)])


        # 3.3) Restore parameter qbits

        inversed(characterise_loop)(*basic_args_to_call_append,k_left,False,True)

    for k_left in range(length-1-min_length_hairpin_loop,length):
        # 3.2 again) Check ( matching
        circuit.append(incr_incorrect.control(nb_bits_bracket_dot,ctrl_state=
    bracket_dot_to_ctrl_state("(")),[folding[k_left][i] for i in range(nb_bits_bracket_dot)
    ]+[incorrect_configuration[i] for i in range(nb_bits_incorrect)])


    # 4) Check <-.-> choice

    #|.->. and .<-.| are prohibited
    for k_dot in range(1,length-1):
        #if 0 == folding[k_dot][0] it is a .
        #then
        # - if 1 == folding[k_dot][1] it is a .->
        #   then
        #   (a) 1 == folding[k_dot-1][0] it is a |
        #   and
        #   (b) 0 == folding[k_dot+1][0] it is a .
        #   is prohibited
        #   so by flipping (a) and (b) if 1 == folding[k_dot][1]
        #   then one get
        #   (flipped a) 0 == folding[k_dot-1][0]
        #   and
        #   (flipped b) 1 == folding[k_dot+1][0]
        #   is prohibited
        #   that is the same condition as the other case since
        #   (flipped a) == (c)
        #   and
        #   (flipped b) == (d)
        #
        # - if 0 == folding[k_dot][1] it is a <-.
        #   then
        #   (c) 0 == folding[k_dot-1][0] it is a .
        #   and
        #   (d) 1 == folding[k_dot+1][0] it is a |
        #   is prohibited

        #flip
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])

        #test
        circuit.append(incr_incorrect.control(3*(nb_bits_bracket_dot-1),ctrl_state=
    bracket_dot_to_ctrl_state("|")+bracket_dot_to_ctrl_state(".")+bracket_dot_to_ctrl_state(
    ".")),[folding[k_dot-1][0]]+[folding[k_dot][0]]+[folding[k_dot+1][0]]+[
```

```
    incorrect_configuration[i] for i in range(nb_bits_incorrect)])

        #flip back
        circuit.cx(folding[k_dot][1],folding[k_dot-1][0])
        circuit.cx(folding[k_dot][1],folding[k_dot+1][0])


    return "check folding"


length = 5

_,state,_ = simulate(check_folding,length)

for (s,_) in state:
    print(s)
    print("\n")
```

Listing 15 – **`characterise_loop(compute_energy=false)`** works. It aims to check that braces are well parenthesised.

B Interactive computing version of the code

The code is organised in the file `oracle.ipynb` attached to this report. It can be read with several interactive computing tools such as Jupyter Notebook (`https://jupyter.org/`) which can be installed, but also exist in a browser demo version (`https://jupyter.org/try`).

C Abstract/Résumé

English abstract Biological linear polymers are concerned by tow main bioinformatics problems. First, the folding problem, that consists in predicting the folding structure from the sequence of monomers. Second, the inverse folding problem, consists in designing a sequence that would fold into a target structure.

The folding and inverse folding problems of RNA are subject to much less attention than their counterpart about proteins. Quantum computing approaches to these problems are no exception and the present work is, as far as we know, the first to propose a quantum algorithm for RNA design.

The Grover search algorithm family provide a favorable framework for solving exactly that kinnd of combinatorial optimisation problems. Here we propose a quantum algorithm dedicated to serve as an oracle in a Grover search derived algorithm. Based on a simplified version of the nearest neighbour model of the RNA secondary structure free energy, the proposed algorithm is simple enough so that key parts of it can be easily simulated, and yet stays of biological relevance while tackling computational difficulties of the problem that classical algorithm struggle to deal with.

The automaton strategy adopted for the algorithm make it quite flexible, and so it should be easy to adapt to other RNA folding models.

Résumé en français Concernant les polymers linéaires biologiques, deux problèmes bioinformatiques principaux se posent. Premièrement le problème du repliment, qui à partir de la séquence consiste à prédire la structure du repliement. Deuxièmement le problème inverse du repliment, qui à partir d'une structure cible de repliement consiste à construire une séquence qui adoptera cette structure, c'est-à-dire le problème du design.

Les problèmes du repliment et du design d'ARN font l'objet de moins d'attention que les mêmes problèmes concernant les protéines. Les approche s'appuyant sur le calcul quantique ne font pas exception et le présent travail est à notre connaissance le premier à proposer un algorithme quantique pour le design d'ARN.

Les algorithmes dérivés de l'algorithme de Grover offrent un cadre de travail propice à la résolution exacte de ce type de problème d'optimisation combinatoire. Il s'agit ici de proposer un oracle pour une approche de type recherche de Grover. En se basant sur le nearest neighbour model de l'énergie libre associée aux structures secondaires de l'ARN, on propose un algorithme quantique qui est suffisamment simple pour pouvoir en simuler aisément des parties clefs et qui pour autant demeure biologiquement pertinent tout en n'évitant pas les difficultés calculatoires qui posent problème aux algorithmes classiques.

La stratégie de type automate de l'algorithme le rend flexible et donc adaptables à d'autres modèles du repliement de l'ARN.