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

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

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

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

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

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

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

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

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

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

        else:
            value += line

        if (set(value).issubset(NUCS) and
                len(value) >= 20):
```

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

```python
            dict[key] = value[:25]
    return dict

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

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

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

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

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

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

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

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

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

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

    return func

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

    if seq1 == 'sensor':
        L = 19

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

    if func > 1:
        func = 1
    return func

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

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

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

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

```
            + seqs[seq2]))
        seq_DG[name] = mfe
        seq_ss[name] = (ss[: len(seqs[seq1])]
            + '&'
            + ss[(len(seqs[seq1])) :-1])

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

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

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

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

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

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

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

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

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

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

    return seqs_aftermutation

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

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

        if stand < 0.1:
            continue

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

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

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

        name = '_'.join(name)

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

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

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

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

```

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

```
    eq_1 = NuPACK(
        Sequence_List=seq_list,
        material='dna')
    eq_2 = NuPACK(
        Sequence_List=seq_list,
        material='dna')

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

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

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

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

    for list in guides:
        for el in list:

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

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

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

    data = [trace1, trace2]
    layout = go.Layout(
        barmode='group',
```

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

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

    return None

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        if i == 1000:
            break

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

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

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

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

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

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

    return outdict, keyss

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

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

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

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

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

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

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

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

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

    OUTFILE.write('\nP1 = ' + str(Dats_preit[0]) + '\n')
```

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

```
    OUTFILE.write('P2 = ' + str(Dats_preit[1]) + '\n')
    OUTFILE.write('P3 = ' + str(Dats_preit[2]) + '\n')
    OUTFILE.write('P4 = ' + str(Dats_preit[3]) + '\n')
    OUTFILE.write('P5 = ' + str(Dats_preit[4]) + '\n')
    OUTFILE.write('Toehold = ' + str(Dats_preit[5]) + '\n')
    OUTFILE.write('Score = ' + str(Score_preit) + '\n')
    OUTFILE.write('Standarized score = '
        + str(Score_preit*100/Dats_preit[3])
        + '\n')


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

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

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

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

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

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

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

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

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

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

k = 0
perc_0 = 0

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

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

    else:
        Metropolis()

    progress()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                + wo_fuel[el][0]
                + '\n'
                + str(wo_fuel[el][1])
                + '\n')


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

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

seqs_preit = {}

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

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

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

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

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

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

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

print('Job finished on '
    + str(datetime.datetime.now()).split('.')[0]
    + '. Elapsed time was: '
    + elapsed_time[:-13]
    + ' minutes.')
```

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

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

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

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

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

#The Ribosome Binding Site Calculator is free software: you can redistribute it
and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.

#The Ribosome Binding Site Calculator is distributed in the hope that it will be
 useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.

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

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

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

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

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

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

debug = 0

#Class that encapsulates all of the functions from NuPACK 2.0


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

    def __init__(self, Sequence_List, material):

        self.ran = 0

        import re
        import string

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

        self._write_input_concentrations(concentrations)

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

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

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

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


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

        self["mfe_composition"] = strands

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

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

        #Write input files
        self._write_input_mfe(strands)

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

        #Call NuPACK C programs
        cmd = nupackbin_dir + "mfe"
```
Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

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

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

        self["subopt_composition"] = strands

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

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

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

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

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

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

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

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

        self["energy_composition"] = strands

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

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

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

        #Set arguments
        material = self["material"]
        if multi == "": multi = ""
        if pseudo: pseudo = " -pseudo"
        if degenerate: degenerate = " -degenerate "
        dangles = " -dangles " + dangles + " "
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

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

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

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

        return energy

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

        self["pfunc_composition"] = strands

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

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

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

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

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

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

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

        file = open(str(self.prefix) + ".func")
        lectura = file.readlines()
        inf = []
        for line in lectura:
            line = line.strip("\n")
            if line[0] != "%" and line[0] != "Attempting":
                inf.append(float(line))
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        file.close()

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

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

        return partition_function

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

        self["count_composition"] = strands

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

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

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

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

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

                                 ".count", shell=True)

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


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

        return number

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

    def _write_input_concentrations(self, concentrations):

        handle = open(self.prefix + ".con", "w")
```
Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

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

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

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

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

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

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

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

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

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

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

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

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

        NumEachStrands = ""
        for num in strands:
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

        input_str = input_str + NumEachStrands + "\n"

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

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

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

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

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

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

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

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

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

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

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

        line = handle.readline()

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

            words=line.split()

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

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

            line = handle.readline()

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

        while line:
            words = line.split()

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

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

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

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

    def _read_output_ocx(self):

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

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

        line = handle.readline()

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

            words = line.split()

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

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

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

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

            line = handle.readline()

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

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

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

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

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


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

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

        #Skip the comments of the text file.

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

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

        while line:
            words = line.split()

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


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

                self["ordered_totalnt"].append(totalnt)

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

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

                self["ordered_energy"].append(mfe)

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

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

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

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

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

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



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

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

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

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

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

```
        bp_y = []

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

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


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

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

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

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

        counter = 0

        while line:
            words = line.split()

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


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

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

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

                self["subopt_energy"].append(mfe)

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

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

                line = handle.readline()
                words = line.split()
                while not line == "\n" and not words[0] == "%":
                    bp_x.append(int(words[0]))
                    bp_y.append(int(words[1]))
                    words = handle.readline().split()
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

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

        self["subopt_NumStructs"] = counter

    def _cleanup(self, suffix):

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

        return

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

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

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


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

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

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

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

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

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

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

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

        #hacksy way of reading from data produced by 'complex', by 'mfe', or by
'subopt'
        data_x = program + "_basepairing_x"
        data_y = program + "_basepairing_y"
        mfe_name = program + "_energy"
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

```
        composition_name = program + "_composition"

        #Format of .ct file

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

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

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


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

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

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

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

        seq_len = len(allseq)

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


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

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

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

        #Write the header
        handle.write(header)

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


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

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

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

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

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

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

            #Init
            hc_5p = 0
            hc_3p = 0

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

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

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

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

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

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

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

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

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

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


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

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

            handle.write(line)

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


if __name__ == "__main__":

    import re

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


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

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

    print(test)

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

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

        output = output + "  dG (RT ln Q): " + \
                 str(test["ordered_energy"][counter]) + " kcal/mol"
        output = output + "  # Permutations: " + \
                 str(test["ordered_permutations"][counter])
```
Annex II: *Code of the Nupack wrapper (Salis et al., 2009) (continues on the next page)*

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

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

    #test.mfe([1, 2], dangles = "all")
    #num_complexes = test["mfe_NumStructs"]  #Number of degenerate complexes (sa
me energy)
    #dG_mfe = test["mfe_energy"]
    #print "There are ", num_complexes, " configuration(s) with a minimum free e
nergy of ", dG_mfe, " kcal/mol.
```
***Annex II**: Code of the Nupack wrapper (Salis et al., 2009)*