SAUNA (Python), code from SAUNA.py	NucPosSimulator (C++) relevant code for comparison pasted from the corresponding source files. Code obtained from: Robert Schöpflin, Vladimir B. Teif, Oliver Müller, Christin Weinberg, Karsten Rippe and Gero Wedemann: Modeling nucleosome position distributions from experimental nucleosome positioning maps. Bioinformatics, 29 (19), 2380-2386, 2013. Through: https://bioinformatics.hochschule-stralsund.de/nucpos/download.html
Legends: Modified Added Deleted Translated into python, functionally identical	

import types
import gc
import tempfile
import os.path as path
import sys
import os
import pandas as pd
import ctypes
import shutil
import inspect
import random
import math
from typing import List, Iterator, Optional, Tuple, Union
import numpy as np
import gzip
 	
EPS = 2.220446049250313e-16
K_B = 8.314513e-3 # in kJ/(mol * K)   GROMACS units
REFERENCE_TEMPERATURE = 293.0 # K
GENERIC_NUC_LENGTH = 147 # bp
MIN_PROBABILITY = 0.000000001
 	
# Move constants 	
MAX_NUC_SHIFT = 60 # bp
MAX_NUC_PAIR_SHIFT = 60 # bp
	const double EPS = numeric_limits<double>::epsilon();
	const double K_B = 8.314513e-3; // in kJ/(mol * K) GROMACS units
	const double REFERENCE_TEMPERATURE = 293.0; // K
	const int GENERIC_NUC_LENGTH = 147; // bp
	const double MIN_PROBABILITY = 0.000000001;
	
	// move constants
	const long int MAX_NUC_SHIFT = 50; // bp
	const long int MAX_NUC_PAIR_SHIFT = 20; // bp

ADD_RATE = 4*10**(-6)
DELETE_RATE = 4*10**(-6)
SHIFT_RATE = 0.55 - 4*10**(-6)
PAIR_SHIFT_RATE = 0.45-4*10**(-6)
We changed the parameters here, because SAUNA starts with an existing nucleosome configuration, whereas the NucPosSimulator starts with naked DNA. Thus, the add and delete rates are low, because we are already starting with roughly the right number of nucleosomes.
# IO constants 	
MAX_LOCUS_LENGTH = 10_000_000_000 # bp
The maximum locus length was changed because to allow for whole genome analyses
	const double ADD_RATE = 0.2;
	const double DELETE_RATE = 0.2;
	const double SHIFT_RATE = 0.4;
	const double PAIR_SHIFT_RATE = 0.2;
	
	// IO constants
	const long int MAX_NUM_OF_READS = 10000000;// nucleosome reads
	We deleted this because SAUNA does not receive reads as input, it takes the already calculated nucleosome occupancy values as input
	const long int MAX_LOCUS_LENGTH = 10000000;// bp

def process_file(file_location):
    # Check if the file exists
    if not os.path.exists(file_location):
        print("File not found!")
        return
 	
    df = pd.read_csv(file_location, sep='\t', usecols=[1], header = None)
    df = df.values.flatten()
    return df
Reading in input nucleosome configuration	

class AbstractException(BaseException):
    def __init__(self, msg, file, line):
        self.msg = msg
        self.file = file
        self.line = line
 	
    def getMessage(self):
        return self.msg
 	
    def getFile(self):
        return self.file
	AbstractException::AbstractException(string msg, string file, int line)
	: msg(msg), file(file), line(line) {
	}
	
	AbstractException::~AbstractException() throw() {
	}
	
	string AbstractException::getMessage() {
	    return msg;
	}
	
	string AbstractException::getFile() {

def getLine(self):
      return self.line
	return file;
	}
	
	int AbstractException::getLine() {
	   return line;
	}

class NucPosRunTimeException(AbstractException):
   def __init__(self, msg, file, line):
      super().__init__(msg, file, line)
 	
   def __del__(self):
      pass # No special cleanup needed in Python
	NucPosRunTimeException::NucPosRunTimeException(string msg, string file, int line)
	: AbstractException(msg,file,line) {
	}
	
	NucPosRunTimeException::~NucPosRunTimeException() throw() {
	}

class NucPosIOException(AbstractException):
   def __init__(self, msg, file, line):
      super().__init__(msg, file, line)
 	
   def __str__(self):
      return f"NucPosIOException: {self.msg} at {self.file}:{self.line}"
 	
   def __repr__(self):
      return f"NucPosIOException('{self.msg}', '{self.file}', {self.line})"
	NucPosIOException::NucPosIOException(string msg, string file, int line)
	: AbstractException(msg,file,line) {
	}
	
	NucPosIOException::~NucPosIOException() throw() {
	}

N = 624
M = 397
MATRIX_A = 0x9908b0df   # constant vector a
UPPER_MASK = 0x80000000 # most significant w-r bits
LOWER_MASK = 0x7fffffff # least significant r bits
 	
mt = (ctypes.c_uint32 * N)()
mti = N + 1 # mti==N+1 means mt[N] is not initialized
 	
def init_genrand(s):
   global mt, mti
	/* Period parameters */
	#define N 624
	#define M 397
	#define MATRIX_A 0x9908b0dfUL /* constant vector a */
	#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
	#define LOWER_MASK 0x7fffffffUL /* least significant r bits */
	
	static unsigned long mt[N]; /* the array for the state vector */
	static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
	
	/* initializes mt[N] with a seed */

```
    mt[0] = s & 0xffffffff
    for i in range(1, N):
        mt[i] = (1812433253 * (mt[i-1] ^ (mt[i-1] >> 30)) + i) & 0xffffffff
    mti = N

def init_by_array(init_key, key_length):
    global mt, mti
    init_genrand(19650218)
    i, j = 1, 0
    k = max(N, key_length)
    while k:
        mt[i] = ((mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525)) +
                  init_key[j] + j) & 0xffffffff
        i += 1
        j += 1
        if i >= N:
            mt[0] = mt[N-1]
            i = 1
        if j >= key_length:
            j = 0
        k -= 1
    for k in range(N-1, 0, -1):
        mt[i] = ((mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941)) - i) & 0xffffffff
        i += 1
        if i >= N:
            mt[0] = mt[N-1]
            i = 1
    mt[0] = 0x80000000  # MSB is 1; assuring non-zero initial array

def genrand_int32():
    global mt, mti
    mag01 = [0x0, MATRIX_A]
    if mti >= N:
        if mti == N + 1:
            init_genrand(5489)
```

```
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
        (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
/* slight change for C++, 2004/2/26 */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
        for kk in range(N-M):
            y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK)
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1]
        for kk in range(N-M, N-1):
            y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK)
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1]
        y = (mt[N-1] & UPPER_MASK) | (mt[0] & LOWER_MASK)
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1]
        mti = 0
    y = mt[mti]
    mti += 1
    y ^= (y >> 11)
    y ^= (y << 7) & 0x9d2c5680
    y ^= (y << 15) & 0xefc60000
    y ^= (y >> 18)
    return y

def genrand_int31():
    return genrand_int32() >> 1

def genrand_real1():
    return genrand_int32() * (1.0 / 4294967295.0)

def genrand_real2():
    return genrand_int32() * (1.0 / 4294967296.0)

def genrand_real3():
    return ((genrand_int32() >> 1) + 0.5) * (1.0 / 4294967296.0)

def genrand_res53():
    a = genrand_int32() >> 5
    b = genrand_int32() >> 6
    return (a * 67108864.0 + b) * (1.0 / 9007199254740992.0)
```

```
          - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0,0xffffffff]-interval */
unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }
    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* generates a random number on [0,0x7fffffff]-interval */
long genrand_int31(void)
{
    return (long)(genrand_int32()>>1);
}

/* generates a random number on [0,1]-real-interval */
double genrand_real1(void)
{
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void)
{
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void)
{
```

<table>
<tr>
<td></td>
<td>

```
   return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
   /* divided by 2^32 */
}

/* generates a random number on [0,1) with 53-bit resolution*/
double genrand_res53(void)
{
   unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
   return(a*67108864.0+b)*(1.0/9007199254740992.0);
}
/* These real versions are due to Isaku Wada, 2002/01/09 added */
```

</td>
</tr>
<tr>
<td>

```
class Interval:
# Define the TYPE enumeration class TYPE: DNA = "DNA" NUC = "NUC"
   def __init__(self, begin: int, end: int, type_: str):
      assert end >= begin, "End must be greater than or equal to begin"
      self.begin = begin
      self.end = end
      self.length = end - begin
      self.type_ = type_

   def getBegin(self) -> int:
      return self.begin

   def getEnd(self) -> int:
      return self.end

   def getLength(self) -> int:
      return self.length

   def getType(self) -> str:
      return self.type_

   def isInInterval(self, position: int) -> bool:
```

</td>
<td>

```
Interval::Interval(long int begin, long int end, TYPE type)
:begin(begin), end(end), length(end-begin), type(type){
   assert(end>=begin);
}

Interval::~Interval() {
}

long int Interval::getBegin() const {
   return begin;
}

long int Interval::getEnd() const {
    return end;
}

long int Interval::getLength() const {
   return length;
}

Interval::TYPE Interval::getType() const {
   return type;
```

</td>
</tr>
</table>

```python
        return self.begin <= position < self.end

    def setValues(self, begin: int, end: int):
        assert end > begin, "End must be greater than begin"
        self.begin = begin
        self.end = end
        self.length = end - begin
```

```cpp
}

bool Interval::isInInterval(long int position) {
    if(begin <= position && position < end) {
        return true;
    }
    return false;
}

void Interval::setValues(long int begin, long int end) {
    assert(end>begin);
    this->begin = begin;
    this->end   = end;
    this->length= end-begin;
}
```

```python
class Configuration:
    def init(self, filename: str, start_nucs_input: np.ndarray, minVal: int, length:
int,locusBegin: int, locusLength: int, nucLength: int, chromosome: str):
        self.nucLength = nucLength
        self.chromosome = chromosome

            assert nucLength >= GENERIC_NUC_LENGTH  # bp
            assert locusBegin >= 0
            assert locusLength > 0

            self.locusBegin = locusBegin
            self.locusLength = locusLength

            start_nucs_input = start_nucs_input-minVal
            start_nucs_input += 147
            start_nucs_input = filter_peak_positions(start_nucs_input, 147)
Here we use the peaks from another peak caller as input to get the starting
nucleosome positions
        self.intervals,self.numOfNucleosomes =
```

```cpp
Configuration::Configuration(long int locusBegin, long int locusLength,
                             long int nucLength, string chromosome)
 : nucLength(nucLength), chromosome(chromosome) {

    assert(nucLength >= GENERIC_NUC_LENGTH); // bp
    assert(locusBegin >= 0);
    assert(locusLength  > 0);

    this->locusBegin = locusBegin;
    this->locusLength  = locusLength;

    intervals.push_back(Interval(0, locusLength, Interval::DNA));
    numOfNucleosomes = 0;
Deleted because SAUNA starts with a nucleosome configuration
    energy          = 0;
    step            = 0;
    temperature     = 0;
}
```

```
get_start_Intervals_new(nucLength,locusLength, start_nucs_input)
```

Here we need to calculate the starting intervals based on the nucleosome positions that we get in the input.

```
        del start_nucs_input
        gc.collect()
```

To reduce memory usage

```
        self.energy = 0
        self.step = 0
        self.temperature = 0

    def getInterval(self, position):
        index = self.getIntervalIter(position)
        return self.intervals[index]
```

Here we define index as the index and not the interval itself when using the getIntervalIter function. Thus, we need to retrieve the interval afterwards. The fact that this returns just the index instead of an iterator is a minor difference, but affects how the code below functions. It should increase its speed.

```
    def getIntervalIter(self, position):
        low = 0
        high = len(self.intervals) - 1
        result = high  # Initialize result to the last interval index

        while low <= high:
            mid = (low + high) // 2
            interval = self.intervals[mid]

            if interval.isInInterval(position):
                result = mid
                break
            elif interval.getEnd() <= position:
                low = mid + 1
            else:
                high = mid - 1
```

```
Configuration::~Configuration() {
}

Interval Configuration::getInterval(long int position) {
    list<Interval>::iterator it =  getIntervalIter(position);
    return (*it);
}

list<Interval>::iterator Configuration::getIntervalIter(long int position) {
    list<Interval>::iterator result = intervals.end();
    bool found = false;
    for (list<Interval>::iterator it = intervals.begin();
        it != intervals.end();
        it++) {
        if ((*it).isInInterval(position)) {
            result  = it;
            found = true;
            break;
        }
    }
    assert(found == true);
    return result;
}
```

This performs a linear search, which is inefficient for large datasets

```python
        assert self.intervals[result].isInInterval(position)  # Ensure that an interval was found
        return result
```

Binary search increases efficiency significantly, assuming that the intervals are sorted and not overlapping (as they should be)

Also: function here returns an index of the interval, whereas the function in the C++ code returns an iterator pointing towards the interval

```python
    def addNucleosome(self, startPosition: int):
        it = self.getIntervalIter(startPosition)

        assert self.intervals[it].getType() == Interval.TYPE.DNA
        assert self.intervals[it].getEnd() > startPosition + self.nucLength
        assert self.intervals[it].isInInterval(startPosition)

        begin = self.intervals[it].getBegin()
        end = self.intervals[it].getEnd()
```

Here, “it” refers to the index of the interval, whereas it refers to the iterator pointing towards the interval in the C++ code

```python
        b1 = begin
        e1 = startPosition
        b2 = startPosition
        e2 = startPosition + self.nucLength
        b3 = startPosition + self.nucLength
        e3 = end

        self.intervals[it].setValues(b1, e1)  # Set new boundaries of DNA interval
```

See above

```python
        it += 1  # Step forward
        self.intervals = np.insert(self.intervals, it, Interval(b2, e2, Interval.TYPE.NUC), axis=0)

        self.numOfNucleosomes += 1
```

```cpp
void Configuration::addNucleosome(long int startPosition) {
    list<Interval>::iterator it = getIntervalIter(startPosition);

    assert((*it).getType() == Interval::DNA);
    assert((*it).getEnd() > startPosition+nucLength);
    assert((*it).isInInterval(startPosition));

    long int begin = (*it).getBegin();
    long int end   = (*it).getEnd();

    long int b1 = begin;
    long int e1 = startPosition;
    long int b2 = startPosition;
    long int e2 = startPosition+nucLength;
    long int b3 = startPosition+nucLength;
    long int e3 = end;

    (*it).setValues(b1,e1); // set new boundaries of DNA interval
    it++; // step on further
    intervals.insert(it, Interval(b2,e2,Interval::NUC));
    numOfNucleosomes++;
    intervals.insert(it, Interval(b3,e3,Interval::DNA));

    assert(b1 < e1);
    assert(e1 == b2);
    assert(b2 < e2);
    assert(e2 == b3);
```

```
        self.intervals = np.insert(self.intervals, it+1, Interval(b3, e3, Interval.TYPE.DNA), axis=0)
```

self.interval is redefined when a nucleosome is added, due to use of numpy array instead of list

```
        assert b1 < e1
        assert e1 == b2
        assert b2 < e2
        assert e2 == b3
        assert b3 < e3


    def deleteNucleosome(self, index: int):
        assert 0 <= index < self.numOfNucleosomes
        it = self.getNucleosomeIter(index)

        assert it != 0  # Ensure it's not the first element
        it -= 1
        assert self.intervals[it].getType() == Interval.TYPE.DNA
        dna0 = self.intervals[it]
        it += 1
        assert self.intervals[it].getType() == Interval.TYPE.NUC
        nuc = self.intervals[it]
        nuc_index = it
        it += 1
        assert self.intervals[it].getType() == Interval.TYPE.DNA
        dna1 = self.intervals[it]
```

“it” refers to index here and not iterator

```
        it += 1  # Move one position beyond the linker DNA

        dna0.setValues(dna0.getBegin(), dna1.getEnd())

        self.intervals = np.delete(self.intervals, slice(nuc_index, it), axis=0)
```

intervals is redefined due to using numpy array instead of list

```
    assert(b3 < e3);
}

void Configuration::deleteNucleosome(long int index) {
    assert(0 <= index  && index < numOfNucleosomes);

    list<Interval>::iterator it = getNucleosomeIter(index);
    assert(it != intervals.begin());
    it--;
    assert((*it).getType() == Interval::DNA);
    list<Interval>::iterator dna0 = it;
    it++;
    assert((*it).getType() == Interval::NUC);
    list<Interval>::iterator nuc  = it;
    it++;
    assert((*it).getType() == Interval::DNA);
    list<Interval>::iterator dna1 = it;
    assert(it != intervals.end());
    it++; // one behind the linker dna
    (*dna0).setValues((*dna0).getBegin(), (*dna1).getEnd());
    // erase the nucleosome and the linker dna behind
    intervals.erase(nuc, it);
    numOfNucleosomes--;
}

list<Interval>::iterator Configuration::getNucleosomeIter(long int nucIndex) {
    assert(0 <= nucIndex  && nucIndex < numOfNucleosomes);
    list<Interval>::iterator result = intervals.begin();
    int count=-1;
    bool success=false;
    for (list<Interval>::iterator it = intervals.begin();
        it != intervals.end();
        it++) {
        if ((*it).getType() == Interval::NUC) {
            count++;
        self.numOfNucleosomes -= 1

    def getNucleosomeIter(self, nucIndex: int):
        assert 0 <= nucIndex < self.numOfNucleosomes
        index = int((nucIndex+1)*2 -1)

        # Ensure that the nth nucleosome is found
        assert index != -1, f"Nucleosome with index {nucIndex} not found"
        return index
```

Here, the function returns the index of the interval, whereas the C++ function returns an iterator.

Importantly, the logic of the search is different in both cases: Here we assume the intervals are always switching between ‘NUC’ and ‘DNA’. I.e. the intervals start with ‘DNA’ and end with ‘DNA and there are never two ‘DNA’ intervals next to each other or two ‘NUC’ intervals next to each other. Thus, we calculate the index of the nth nucleosome directly by using its number (n). This makes the search significantly faster.

```
            if (count == nucIndex) {
                result=it;
                success=true;
                break;
            }
        }
    }
    assert(success);
    return result;
}
```

```
    def shiftNucleosome(self, index: int, distance: int):
        assert 0 <= index < self.numOfNucleosomes
        if distance == 0:
            return

        it = self.getNucleosomeIter(index)

        assert it != 0  # Ensure it's not the first element
```

index vs. iterator, see above

```
        it -= 1
        assert self.intervals[it].getType() == Interval.TYPE.DNA
        dna0 = self.intervals[it]

        it += 1
        assert self.intervals[it].getType() == Interval.TYPE.NUC
        nuc = self.intervals[it]
```

```
void Configuration::shiftNucleosome(long int index, long int distance ) {
    assert(0 <= index  && index < numOfNucleosomes);
    if (distance == 0) {
        return;
    }

    list<Interval>::iterator it = getNucleosomeIter(index);
    assert(it != intervals.begin());
    it--;
    assert((*it).getType() == Interval::DNA);
    Interval &dna0 = (*it);
    it++;
    assert((*it).getType() == Interval::NUC);
    Interval &nuc  = (*it);
    it++;
    assert(it != intervals.end());
    assert((*it).getType() == Interval::DNA);
```

```
        assert it != len(self.intervals)-1  # Ensure it's not the end of the list
        it += 1
        assert self.intervals[it].getType() == Interval.TYPE.DNA
        dna1 = self.intervals[it]

    # Shift to the left
        if distance < 0:
            assert dna0.getEnd() - dna0.getBegin() > (-1 * distance)

        else:  # Shift to the right
            assert dna1.getEnd() - dna1.getBegin() > distance



        b1 = dna0.getBegin()
        e1 = dna0.getEnd() + distance
        b2 = nuc.getBegin() + distance
        e2 = nuc.getEnd() + distance
        b3 = dna1.getBegin() + distance
        e3 = dna1.getEnd()
        dna0.setValues(b1, e1)
        nuc.setValues(b2, e2)
        dna1.setValues(b3, e3)
```

```
    Interval &dna1 = (*it);

    // shift to the left
    if (distance < 0) {
        assert(dna0.getEnd()-dna0.getBegin() > (-1*distance) );
    } else { // shift to the rigth
        assert(dna1.getEnd()-dna1.getBegin() > (distance) );
    }

    long int b1 = dna0.getBegin();
    long int e1 = dna0.getEnd()   + distance;
    long int b2 = nuc.getBegin()  + distance;
    long int e2 = nuc.getEnd()    + distance;
    long int b3 = dna1.getBegin() + distance;
    long int e3 = dna1.getEnd();

    dna0.setValues(b1,e1);
    nuc.setValues( b2,e2);
    dna1.setValues(b3,e3);
}
```

```
    def getNucleosomeInterval(self, nucIndex: int) -> Interval:
        it = self.getNucleosomeIter(nucIndex)
        return self.intervals[it]

    def getStartPositionOfNuc(self, nucIndex: int) -> int:
        assert 0 <= nucIndex < self.numOfNucleosomes
        index = self.getNucleosomeIter(nucIndex)
        it = self.intervals[index]
```

extra step to get interval from index

```
        return it.getBegin()
```

Here, ‘it’ refers to the interval, whereas it refers to the iterator in the C++ code

```
Interval Configuration::getNucleosomeInterval(long int nucIndex) {
    list<Interval>::const_iterator it = getNucleosomeIter(nucIndex);
    return (*it);
}

long int Configuration::getStartPositionOfNuc(long int nucIndex) {
    assert(0<=nucIndex && nucIndex < numOfNucleosomes);
    list<Interval>::const_iterator it = getNucleosomeIter(nucIndex);
    return (*it).getBegin();
}

bool Configuration::isStartPositionFree(long int pos) {
    def isStartPositionFree(self, pos: int) -> bool:
        end = self.intervals[-1].getEnd()

        result = False

        assert 0 <= pos < end
        it = self.getIntervalIter(pos)
        interval = self.getInterval(it)
Extra step to get interval

        if interval.getType() == Interval.TYPE.DNA and interval.getBegin() < pos and interval.getEnd() - self.nucLength > pos:
            result = True
        return result
```

```
    long int end = intervals.back().getEnd();
    assert(0<=pos && pos<end);
    list<Interval>::const_iterator it = getIntervalIter(pos);

    bool result = false;
    if((*it).getType() == Interval::DNA &&
       (*it).getBegin()< pos &&
       (*it).getEnd()-Configuration::nucLength > pos) {
        result = true;
    }
    return result;
}
```

```
    def canShiftNucleosome(self, nucIndex: int, distance: int) -> bool:
        assert 0 <= nucIndex < self.numOfNucleosomes

        result = True
        it = self.getNucleosomeIter(nucIndex)
        assert self.intervals[it].getType() == Interval.TYPE.NUC
        nuc = self.intervals[it]

        if distance < 0:  # Test left shift
            if it == 0:
                result = False
Here we allow the first interval to be chosen without error and return false
            else:
                it -= 1
                assert self.intervals[it].getType() == Interval.TYPE.DNA
                dna = self.intervals[it]
                if nuc.getBegin() + distance <= dna.getBegin():
                    result = False
        elif distance > 0:  # Test right shift
```

```
bool Configuration::canShiftNucleosome(long int nucIndex, long int distance) {
    assert(0 <= nucIndex && nucIndex < numOfNucleosomes);

    bool result = true;
    list<Interval>::iterator it = getNucleosomeIter(nucIndex);
    assert((*it).getType() == Interval::NUC);
    Interval nuc = (*it);

    if (distance < 0) { // test left shift
        assert(it != intervals.begin());
        it--;
        assert((*it).getType() == Interval::DNA);
        Interval &dna = (*it);
        if(nuc.getBegin()+distance <= dna.getBegin() ) {
            result = false;
        }
    } else if (distance > 0) { // test rigth shift
        it++;
        assert(it != intervals.end());
        assert((*it).getType() == Interval::DNA);
```

<table>
<tr>
<td>

```
            if it == len(self.intervals)-1:
                result=False
Same as above
            else:
                it += 1

                assert self.intervals[it].getType() == Interval.TYPE.DNA
                dna = self.intervals[it]
                if nuc.getEnd() + distance >= dna.getEnd():
                    result = False

        return result

    def getNumOfFreePositions(self) -> int:
        freePositions = 0
        for interval in self.intervals:
            if interval.getType() == Interval.TYPE.DNA and interval.getLength() >=
self.nucLength + 2:
                freePositions += interval.getLength() - self.nucLength - 2
        return freePositions
```

</td>
<td>

```
        Interval &dna = (*it);
        if( nuc.getEnd()+distance >= dna.getEnd() ) {
            result = false;
        }
    }

    return result;
}

long int Configuration::getNumOfFreePositions() const {
    long int freePositions = 0;
    for (list<Interval>::const_iterator it = intervals.begin();
        it != intervals.end();
        it++) {
        if ((*it).getType() == Interval::DNA && (*it).getLength() >= nucLength+2) {
            freePositions += (*it).getLength()-nucLength-2;
        }
    }
    return freePositions;
}
```

</td>
</tr>
<tr>
<td>

```
    def canShiftNucleosomePair(self, nucIndex0: int, nucIndex1: int, distance: int)
-> bool:
        assert (nucIndex0 == nucIndex1 - 1) or (nucIndex0 ==
self.numOfNucleosomes - 1 and nucIndex1 == 0)
        assert 0 <= nucIndex0 < self.numOfNucleosomes
        assert 0 <= nucIndex1 < self.numOfNucleosomes

        result = False

        if nucIndex0 == nucIndex1 - 1:
            if distance < 0:
                result = self.canShiftNucleosome(nucIndex0, distance)
            elif distance >0:
                result = self.canShiftNucleosome(nucIndex1, distance)
```

</td>
<td>

```
bool Configuration::canShiftNucleosomePair(long int nucIndex0,
                                          long int nucIndex1,
                                          long int distance) {
    assert( nucIndex0 == nucIndex1-1 || // case 1: standard
           (nucIndex0 == numOfNucleosomes-1 && nucIndex1 == 0)); // case 2: wrap around
    assert(0 <= nucIndex0 && nucIndex0 < numOfNucleosomes);
    assert(0 <= nucIndex1 && nucIndex1 < numOfNucleosomes);
    bool result = false;

    // case 1: standard
    if(nucIndex0 == nucIndex1-1) {
        if (distance < 0) {//shift to the left
            result = canShiftNucleosome(nucIndex0, distance);
        } else {   // shift to the right or no shift
```

</td>
</tr>
</table>

```
        elif nucIndex0 == self.numOfNucleosomes - 1 and nucIndex1 == 0:
            result = False
Wrap around disabled for simplicity
        else:
            raise AssertionError("Mismatching indices in PairShiftMove")
        return result
```

```
            result = canShiftNucleosome(nucIndex1, distance);
        }
    } else if (nucIndex0 == numOfNucleosomes-1 && nucIndex1 == 0) { // case 2: wrap around
        bool r0 = canShiftNucleosome(nucIndex0, distance);
        bool r1 = canShiftNucleosome(nucIndex1, distance);
        if (r0 && r1) {
            result = true;
        }
    } else {
        throw new NucPosRunTimeException("Mismatching indices in PairShiftMove",
                __FILE__, __LINE__);
    }
    return result;
}
```

```
    def getNucIndex(self, pos: int) -> int:
        assert 0 <= pos < self.intervals[-1].getEnd()
        interval_index = self.getIntervalIter(pos)
        nucindex = int((interval_index+1)/2 -1 )
        return nucindex
Calculate nucindex assuming alternating structure of ‘DNA’ and NUC’à significantly decreases computation time when compared to iterating through all intervals

    def setStep(self, step: int):
        assert step > 0
        self.step = step

    def getNucLength(self) -> int:
        return self.nucLength

    def getNumOfNucleosomes(self) -> int:
        return self.numOfNucleosomes

    def setTemperature(self, temperature: float):
```

```
long int Configuration::getNucIndex(long int pos) {
    assert(0 <= pos && pos < intervals.back().getEnd());
    long int count = -1;
    bool success=false;
    for (list<Interval>::iterator it = intervals.begin();
        it != intervals.end();
        it++) {

        if ((*it).getType() == Interval::NUC) {
            count++;
            if ( (*it).getBegin() <= pos && pos < (*it).getEnd() ) {
                success=true;
                break;
            }
        }
    }
    assert(success);
    return count;
}
```

self.temperature = temperature
 	
    def increaseSteps(self):
        self.step += 1
 	
    def addDeltaEnergy(self, deltaEnergy):
        self.energy += deltaEnergy
 	
    def getLength(self) -> int:
        return self.intervals[-1].getEnd()
 	
    def getEnergy(self) -> float:
        return self.energy
 	
    def getStep(self):
        return self.step
 	
    def getChromosome(self):
        return self.chromosome
 	
    def getLocusBegin(self):
        return self.locusBegin
 	
    def getTemperature(self):
        return self.temperature
	
	void Configuration::setStep(long int step) {
	    assert(step > 0);
	    this->step = step;

def filter_peak_positions(peak_positions, min_distance):
    while True:
    peak_positions = peak_positions[np.concatenate(([True], np.diff(peak_positions) > min_distance))]
    if np.all(np.diff(peak_positions) > min_distance):
        break
 	
    return peak_positions	

The input peaks can be overlapping because they are found by another peak caller. Here, we make sure that the peaks for the initial configuration used by the simulation is non-overlapping	

def get_start_Intervals_new(nucLength,locusLength, indices):
    half_nucleosome = int(nucLength/2)
    intervals = [Interval(0,indices[0]-half_nucleosome , Interval.TYPE.DNA),Interval(indices[0]-half_nucleosome,indices[0]-half_nucleosome+ nucLength,Interval.TYPE.NUC), Interval(indices[0]-half_nucleosome+ nucLength,indices[1]-half_nucleosome,Interval.TYPE.DNA)]
    numOfNucleosomes = len(indices)
    for number,index in enumerate(indices):
        if number == 0:
            continue
        start = intervals[number*2].getEnd()
        if number != len(indices)-1:
            subintervals = [Interval(start,start + nucLength,Interval.TYPE.NUC),Interval(start + nucLength,indices[number+1]-half_nucleosome,Interval.TYPE.DNA)]
            intervals.extend(subintervals)
        if number == len(indices)-1:
 	
            if index + half_nucleosome >= locusLength:
                begin = intervals[-1][-1].getBegin()
                intervals[-1][-1].setValues(begin, locusLength)
            else:
 	
                subintervals = [Interval(start, start + nucLength, Interval.TYPE.NUC), Interval(start + nucLength, locusLength,Interval.TYPE.DNA)]
                intervals.extend(subintervals)
    return np.array(intervals),numOfNucleosomes
 	
Here we create the initial configuration used by the simulation	

class Energy:
    def init(self, parent_dir: str, filename: str, probabilities: np.ndarray,
	Energy::Energy(const vector &probabilities, long int locusBegin, long locusEnd, double bindingEnergy) {

```
locusBegin: int, locusEnd: int, bindingEnergy: float):
    # Create NumPy array for energy values
    self.locusBegin = locusBegin
    self.locusEnd = locusEnd
    self.bindingEnergy = bindingEnergy
    def get_penalty(positions, data, distance, penalty_scale = 1):
      total_penalty = np.maximum(0, data[positions - distance] -
data[positions]) + np.maximum(0, data[positions + distance] - data[positions])
      decayed_penalty = np.exp(-total_penalty * penalty_scale)
      del total_penalty
gc.collect()
      return decayed_penalty
penalty function to increase probability of placing nucleosomes at peaks
    def calculate_lower_proximity(positions, data, distance, penalty_scale = 1):
      total_penalty = get_penalty(positions[(positions - distance >= 0) &
(positions + distance < len(data))], data, distance, penalty_scale)
      probabilities = np.zeros(len(positions))
      probabilities[(positions - distance >= 0) & (positions + distance <
len(data))] = total_penalty
      del total_penalty
      gc.collect()
      return probabilities
Function to apply penalty function to the array of probabilites
    probabilities =
calculate_lower_proximity(np.array(range(0,len(probabilities))), probabilities,
40,3000) * probabilities
Apply penalty
    probabilities = np.clip(probabilities, MIN_PROBABILITY, None)
    if np.any(probabilities < MIN_PROBABILITY) or np.any(probabilities > 1.0 +
EPS):
      errMsg = f"Incorrect probability value in Energy construction\n"
      errMsg += f"Value in the range [{MIN_PROBABILITY}, 1.0]"
      frame = inspect.currentframe()
      line_number = frame.f_lineno
      raise NucPosRunTimeException("I", __file__, line_number)
```

```
  this->energyValues = vector<double>( probabilities.size() , 0.0);
  this->locusBegin        = locusBegin;
  this->locusEnd          = locusEnd;
  this->bindingEnergy     = bindingEnergy;

  assert(probabilities.size() ==  energyValues.size());
  assert(locusEnd-locusBegin == (long int) energyValues.size());


  for (vector<double>::size_type i=0; i<energyValues.size(); i++) {

    double probability = probabilities[i];

    // zero is set to a minimum probability, because zero can not be
    // handled in log function
    if (probability < MIN_PROBABILITY) {
      probability = MIN_PROBABILITY;
    }
    if(probability < MIN_PROBABILITY || probability > 1.0+EPS) {
      stringstream errMsg;
      errMsg << "Incorrect probability value in Energy construction: ";
      errMsg << probabilities[i] << "\nValue in the range [" << MIN_PROBABILITY << ",1.0]";
      throw new NucPosRunTimeException("I", __FILE__, __LINE__);
    }
    energyValues[i] = -1.0*(log(probability)*K_B*REFERENCE_TEMPERATURE);
  }
}
```

use of vectorized operations for increased efficiency
 	
    # Calculate energy values using vectorized operations
    filename = filename +".energy.dat"
    # Create a temporary directory within the specified parent directory
    self.temp_dir = tempfile.mkdtemp(dir=parent_dir)
    filename = path.join(self.temp_dir, filename)
 	
    self.energyValues = np.memmap(filename,dtype = "float32", mode='w+', shape=probabilities.shape)
    self.filename = filename
    del filename
Use of memmap to reduce memory usage; important for large arrays (and thus, whole-genome analyses)
    self.energyValues[:] = -1.0 * (np.log(probabilities) * K_B * REFERENCE_TEMPERATURE)
Vectorized operations for increased efficiency
    assert len(probabilities) == len(self.energyValues)
    assert locusEnd - locusBegin == len(self.energyValues)
    del probabilities #remove because uses much memory
    gc.collect()
    self.energyValues.flush()
Force memory cleanup
  def cleanup(self):
    # Remove the temporary directory and its contents
    shutil.rmtree(self.temp_dir)
Clean up temporary directory used for memory mapping
  def getEnergy(self, index: int) -> float:
    assert 0 <= index < len(self.energyValues)
    return self.energyValues[index]
 	
  def get_binding_energy(self) -> float:
    return self.bindingEnergy # Simply return the binding energy attribute
 	
  def getShiftEnergyDifference(self, fromCenterPos: int, toCenterPos: int) ->
	Energy::~Energy() { }
	double Energy::getEnergy(long int index) const {
	  assert(0 <= index && index < (long int)energyValues.size());
	  return energyValues[index]; }
	double Energy::getShiftEnergyDifference(long int fromCenterPos, long int toCenterPos) const {

```
float:
        delEnergy = -self.getEnergy(fromCenterPos)
        addEnergy = self.getEnergy(toCenterPos)
        return delEnergy + addEnergy

    def getDeleteEnergyDifference(self, centerPos: int) -> float:
        return -self.getEnergy(centerPos) - self.bindingEnergy

    def getAddEnergyDifference(self, centerPos: int) -> float:
        return self.getEnergy(centerPos) + self.bindingEnergy
    def get_all_energy(self):
        return self.energyValues
```

```
    double delEnergy = -getEnergy(fromCenterPos);
    double addEnergy =  getEnergy(toCenterPos);

    return delEnergy+addEnergy;
}
double Energy::getDeleteEnergyDifference(long int centerPos) const {
    return -getEnergy(centerPos)-bindingEnergy;
}
double Energy::getAddEnergyDifference(long int centerPos) const {
    return getEnergy(centerPos)+bindingEnergy;
}
void Energy::printValues(ostream &out) {
    out << "# values of the energy function for every position of the locus" << endl;
    out << "# position\tenergy-value" << endl;;
    for (long int i=0; i<(long int)energyValues.size(); i++) {
        out << locusBegin+i << "\t" << getEnergy(i) << endl;
    }
}
```

```
def open_file(file_path):
    columns_to_load = [3]
    arrs = []
    chunk_size = 50000
        # Read the first row to get the first element of the first and second columns
    with gzip.open(file_path, 'rt') as f:
        first_row_df = pd.read_csv(f, sep='\t', usecols=[0,1], nrows=1,
header=None)
        chromosome = first_row_df.iloc[0, 0] # Element of the first row, first
column
        start = first_row_df.iloc[0, 1] # Element of the first row, second column
        del first_row_df
    with gzip.open(file_path, 'rt') as f:
        for chunk in pd.read_csv(f, sep='\t', chunksize=chunk_size,
usecols=columns_to_load, header=None):
            chunk[3] = chunk[3].astype(float)
            arrs.append(chunk.values)
```

arrays = np.concatenate(arrs)
    del arrs
    gc.collect()
 	
    with gzip.open(file_path, 'rt') as f:
        last_row_df = pd.read_csv(f, sep='\t', usecols=[1], nrows=1, header=None, skiprows= (len(arrays)-1))
        end = last_row_df.iloc[0,0] # Last element of the second column
        del last_row_df
 	
    return arrays[:,0], start ,end,chromosome
 	
open the tsv.gz file containing the nucleosome occupancy data (e.g. WPS scores or fragment center coverage) and get the needed occupancy values	

class EnergyFactory:
    def init(self, parent_dir, filename,pReads, locusBegin, locusEnd):
        assert locusBegin >= 0
        assert locusBegin < locusEnd
        assert pReads is not None
        self.filename = filename self.parent_dir = parent_dir
        self.locusBegin = locusBegin # Store locusBegin as an attribute
        self.locusEnd = locusEnd # Store locusEnd as an attribute
        length = locusEnd - locusBegin
        self.nucCenters = np.zeros(length)
numpy to for vectorized operations later on
        del length
        self.nucCenters[147:len(pReads)+147] = pReads
pReads already contains the nucleosome occupancy values (WPS or fragment center coverage)
        del pReads
        gc.collect()
reduce memory usage
	EnergyFactory::EnergyFactory(const vector<pair<long int, long int> > *pReads, long int locusBegin, long int locusEnd) : locusBegin(locusBegin), locusEnd(locusEnd) {
	assert(locusBegin >= 0); assert(locusBegin < locusEnd); assert(pReads != 0);
	long int length = locusEnd - locusBegin;
	nucCenters=vector<long int>(length);
	normSmoothedNucCenters=vector<double>(length);
	
	// collect and accumulate the single reads
	for (vector<pair<long int, long int> >::size_type i = 0; i < pReads->size(); i++) {
	    const long int begin = (*pReads)[i].first;
	    const long int end = (*pReads)[i].second;
	
	    assert(0 <= begin);
	    assert(begin < end);
	
	    long int diff = end-begin;
	    // (long int) ((double)diff/2.0)+0.5) => round values
	    long int index = begin + ((long int) ((double)diff/2.0)+0.5) - locusBegin;
	    assert(0 <=index && index < (long int) nucCenters.size());
	    nucCenters[index]+=1;

```
    def smooth_values(self, sigma):
        # Create the Gaussian kernel
        half_kernel = 8 * np.ceil(sigma)
        kernel_indices = np.arange(-half_kernel, half_kernel + 1)
        kernel = self.gauss(kernel_indices, sigma)
        del kernel_indices
        # Pad nucCenters to handle edge cases
        self.nucCenters = np.pad(self.nucCenters, (int(half_kernel),
int(half_kernel)), mode='edge')
        del half_kernel
        gc.collect()
        # Perform convolution using np.convolve
        self.nucCenters = np.convolve(self.nucCenters, kernel, mode='valid')
        return self.nucCenters
```

limit memory usage

use vectorized operations to improve efficiency

```
}
```

raw reads are processed, rounded midpoints are calculated

```
}
EnergyFactory::~EnergyFactory() { }
vector EnergyFactory::smoothValues(double sigma) { vector nucleosomeCoverage =
vector(nucCenters.size());
int halfKernel = 8*(int)ceil(sigma);
vector<double> kernel(halfKernel*2+1);

// initialize kernel
for (unsigned int i=0; i<kernel.size(); i++) {
   int x = i-halfKernel;
   kernel[i] = gauss((double) x, sigma);
}

// convolution
for (vector<long int>::size_type i=0; i<nucCenters.size(); i++) {

   double value = 0.0;

   for (vector<double>::size_type j=0; j<kernel.size(); j++) {

      long int k = i+j-halfKernel;

      if (0 <= k  && k < (long int)nucCenters.size()) {
         value += kernel[j] * (double)nucCenters[k];
      }
   }

   nucleosomeCoverage[i] = value;
}
return nucleosomeCoverage;

}
```

```
    def gauss(self, x, sigma):
        return 1.0 / (sigma * 2.0 * np.sqrt(2 * np.pi)) * np.exp(-0.5 * (x * x) / (sigma
* sigma))

    def give_energy(self, sigma, binding_energy):
        # Smooth the values
        self.nucCenters = self.smooth_values(sigma)
        # Shift the data to make all values non-negative
        min_value = np.min(self.nucCenters)
        self.nucCenters = self.nucCenters + abs(min_value)
        # Determine max and sum
        maximum = np.max(self.nucCenters)
use of numpy instead of a loop to increase efficiency
        assert maximum > 0
        # Normalize
        self.nucCenters = self.nucCenters / maximum
        assert np.all((0 <= self.nucCenters) & (self.nucCenters <= 1.0))
use vectorized operations to increase efficiency
        return Energy(self.parent_dir,self.filename, self.nucCenters, self.locusBegin,
self.locusEnd, binding_energy)
```

```
double EnergyFactory::gauss(double x, double sigma) const { return
1.0/(sigma2.0sqrt(M_PI_2)) * exp(-0.5*( xx /(sigmasigma))); }
Energy* EnergyFactory::giveEnergy(double sigma, double bindingEnergy) { // smooth the
values vector smoothed_values = smoothValues(sigma);
// determine sum
double sum = 0.0;
Sum is computed but not used

// determine max
double max = numeric_limits<double>::min();
for (vector<double>::size_type i = 0; i < smoothed_values.size(); i++) {

    sum += smoothed_values[i];
    if(smoothed_values[i]>max) {
        max = smoothed_values[i];
    }
}
assert(max > 0);

// normalize
for (vector<double>::size_type i = 0; i < normSmoothedNucCenters.size(); i++) {
    normSmoothedNucCenters[i] = double(smoothed_values[i])/max;
    assert(0 <=normSmoothedNucCenters[i]  && normSmoothedNucCenters[i] <= 1.0);
}

return new Energy(normSmoothedNucCenters, locusBegin, locusEnd,
                bindingEnergy);

}
void EnergyFactory::printFrequencies(ostream &out) { out << "# frequency of nucleosome
centers" << endl; out << "# position\tcount" << endl; for (vector::size_type i=0;
i<nucCenters.size(); i++) { out << locusBegin+i << "\t" << nucCenters[i] << endl; } }
void EnergyFactory::printProbabilities(ostream &out) { out << "# smoothed, relative
occupancy of DNA with nucleosome centers" << endl; out << "# position\trelative-
occupancy" << endl; for (vector::size_type i=0; i<normSmoothedNucCenters.size(); i++) { out << locusBegin+i << "\t" << normSmoothedNucCenters[i] << endl; } }
```

```
class ReadReader:
    def init(self, filename: str):
        self.locusBegin = 0
        self.locusEnd = 0
        min_val = float('inf')
        max_val = float('-inf')
        self.pReads,self.minValue,self.maxValue,chrom = open_file(filename)
use open_file function that was previously defined
        self.chromosome = "chr" + str(chrom)
        del chrom
```

```
ReadReader::ReadReader(const char *filename) {
    pReads =  new vector<pair<long int, long int> >();

    locusBegin = 0;
    locusEnd   = 0;

    long int min = LONG_MAX;
    long int max = LONG_MIN;
    long int numOfReads = 0;

    ifstream inputFile (filename);
    if (inputFile.is_open() == false) {
        stringstream errMsg;
        errMsg << "Unable to open input file: \"" << filename << "\"\n";
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
    }

    const int max_length = 100;
    char chrom[max_length] = {};
    char refchrom[max_length] = {};

    bool first = false;

    long int begin;
    long int end;

    long int lineCount  = 0;
    char line[max_length];

    // fgets returns NULL at EOF
    while( inputFile.good() ) {
        inputFile.getline(line, max_length);
        // skip empty lines
    if (strlen(line) == 0) {
      continue;
    }
    int parsestat = sscanf(line, "%99s%li%li\n", chrom, &begin, &end);
    if (parsestat == 3) {
      if (begin >= end) {
        stringstream errMsg;
        errMsg << "Invalid input from file \"" << filename << "\"\n";
        errMsg << "Nucleosome begin has to be smaller than the end.\n";
        errMsg << "Please check line number: " << lineCount+1;
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
      }
      assert(begin < end);

      if (begin < 0) {
        stringstream errMsg;
        errMsg << "Invalid input from file \"" << filename << "\". \n";
        errMsg << "Invalid begin of read: " << begin << "\n";
        errMsg << "Please check line number: " << lineCount+1;
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
      }

      if (first == false) {
        strncpy(refchrom, chrom, max_length);
        chromosome = string(chrom);
        first = true;
      }

      if (strcmp(chrom, refchrom) != 0) {
        stringstream errMsg;
        errMsg << "Invalid input from file \"" << filename << "\". \n";
        errMsg << "Different chromosomes: " << refchrom << " vs. " << chrom << "\n";
        errMsg << "Please check line number: " << lineCount+1;
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
      }
```

```
            if (begin < min) {
                min = begin;
            }

            if (end > max) {
                max = end;
            }
            pReads->push_back(pair<long int, long int> (begin, end));
            numOfReads++;
            if(numOfReads > MAX_NUM_OF_READS) {
                stringstream errMsg;
                errMsg << "Number of reads reads in file \"" << filename << "\" ";
                errMsg << "exceeds maximum value(" <<  MAX_NUM_OF_READS << ").\n";
                errMsg << "Please reduce the number of reads.";
                throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
            }
        } else {
            stringstream errMsg;
            errMsg << "Skipped input line " << lineCount+1;
            Log::warning(errMsg.str());
        }
        lineCount++;
    }

    inputFile.close();

    // error log
    if (pReads->size() == 0) {
        stringstream errMsg;
        errMsg << "No valid reads in file \"" << filename << ".";
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
    }

    if (max-min > MAX_LOCUS_LENGTH) {
        assert self.minValue >= 0
        assert self.maxValue >= 0

        # add flanking DNA
        self.locusBegin = max(0, self.minValue)
```

```
    stringstream errMsg;
    errMsg << "Length of locus in file \"" << filename << "\" ";
    errMsg << "exceeds maximum value (" << MAX_LOCUS_LENGTH << ").\n" ;
    errMsg << "Please reduce the length of locus by limiting the range of reads." ;
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
  }

  assert(min >= 0);
  assert(max >= 0);

  // add flanking DNA
  locusBegin = min-GENERIC_NUC_LENGTH;
  if(locusBegin < 0) {
    locusBegin = 0;
  }
  locusEnd   = max+GENERIC_NUC_LENGTH;

  assert(pReads->size() > 0);
  assert(locusBegin < locusEnd);
  assert(0<=locusBegin && locusBegin < locusEnd);

  cout << "Imported " << pReads->size() << " nucleosome reads successfully.\n"
     << "Read range:\t" << min << ":" << max << "\n"
     << "Locus begin:\t" << locusBegin << "\n"
     << "Locus end:\t" << locusEnd << endl;
}

ReadReader::~ReadReader() {
}

vector< pair<long int, long int> >* ReadReader::getReads() {
  return pReads;
}

long int ReadReader::getLocusBegin() {
        self.locusEnd = self.maxValue + 2*GENERIC_NUC_LENGTH
add two flanks, shift all values 1 nucleosome length to the right
        assert len(self.pReads) > 0
        assert self.locusBegin < self.locusEnd
        assert 0 <= self.locusBegin < self.locusEnd

        print(f"Imported {len(self.pReads)} nucleosome reads successfully.\n"
          f"Read range:\t{self.minValue}:{self.maxValue}\n"
          f"Locus begin:\t{self.locusBegin}\n"
          f"Locus end:\t{self.locusEnd}")

    def getReads(self) -> List[Tuple[Union[str, int], int, int]]:
        return self.pReads

    def getLocusBegin(self) -> int:
        return self.locusBegin

    def getLocusEnd(self) -> int:
        return self.locusEnd

    def getLocusLength(self) -> int:
        return self.locusEnd - self.locusBegin

    def getChromosome(self) -> str:
        return self.chromosome
    def getMin(self)->int:
        return  self.minValue
    def getMax(self)->int:
        return  self.maxValue
```

```
    return locusBegin;
}

long int ReadReader::getLocusEnd() {
    return locusEnd;
}

long int ReadReader::getLocusLength() {
    return locusEnd-locusBegin;
}

string ReadReader::getChromosome() {
    return chromosome;
}
```

```
class ConfigWriter:
    def init(self): pass
    def writeConfig2Bed(self, config, out):
        for interval in config.intervals:
            if interval.getType() == Interval.TYPE.NUC:
```

```
ConfigWriter::ConfigWriter() {
}

ConfigWriter::~ConfigWriter() {
}
```

```
        out.write(f"{config.getChromosome()}\t"
             f"{config.getLocusBegin() + interval.getBegin()-147}\t"
             f"{config.getLocusBegin() + interval.getEnd()-147}\n")
```

shift nucleosomes, because we have padding on both sides

```
    out.flush()
    del config
    gc.collect()
```

reduce memory usage

```
void ConfigWriter::writeConfig2Bed(const Configuration &config,
                                 ostream &out) {
for(list<Interval>::const_iterator it = config.begin();
it != config.end();
it++) {

if((*it).getType() == Interval::NUC) {
out << config.getChromosome() << "\t"
<< config.getLocusBegin() + (*it).getBegin() << "\t"
<< config.getLocusBegin() + (*it).getEnd() << "\n";
}
}
out.flush();
}

void ConfigWriter::writeConfigAndSimInfo2Bed(const Configuration &config,
                                 ostream &out) {

out << "#SIMINFO: simStep\tchromosome\tlocusBegin\tlocusLength"
<< "\tnucLength\tnumOfNucs\ttemperature\n";

out << "#BEGIN\t" << config.getStep()
<< "\t" << config.getChromosome()
  << "\t" << config.getLocusBegin()
<< "\t" << config.getLength()
  << "\t" << config.getNucLength()
<< "\t" << config.getNumOfNucleosomes()
<< "\t" << config.getTemperature()
<< "\n" ;
writeConfig2Bed(config, out);
out << "#END"<< endl;
}
```

```
class MoveSelector:
  def init(self):
    self.moves = []
    self.cumulated_probabilities = []
  def __del__(self):
    for move in self.moves:
      del move

  def addMove(self, move, probability):
    self.moves.append(move)
    self.cumulated_probabilities.append(probability)
    n = len(self.cumulated_probabilities)
    # Cumulate probabilities
    if n > 1:
      self.cumulated_probabilities[-1] += self.cumulated_probabilities[-2]

  def next_(self):
    # Check that overall probability is 1
    assert math.isclose(self.cumulated_probabilities[-1], 1.0, abs_tol=EPS)

    # Get a random number between 0 and 1
    number = genrand_real1()

    # Select next move
    p_result = None
    for i, probability in enumerate(self.cumulated_probabilities):
      if number <= probability:
        p_result = self.moves[i]
        break
    assert p_result is not None
    return p_result

  def printRates(self):
    print("----------------------------------------\nMove acceptance rates:")
    for move in self.moves:
```

```
MoveSelector::MoveSelector() { }
MoveSelector::~MoveSelector() {
  for(vector::size_type i=0; i<cumulatedProbabilities.size();i++) {
    if(moves[i] != NULL) {
      delete moves[i];
    }
  }
}
void MoveSelector::addMove(AbstractMove *move, double propability) {
  moves.push_back(move);
  cumulatedProbabilities.push_back(propability);
  const vector::size_type n = cumulatedProbabilities.size();
  // probabilities are cumulated
  if(n > 1) {
cumulatedProbabilities[n-1]=cumulatedProbabilities[n-1]+cumulatedProbabilities[n-2]; } }
AbstractMove* MoveSelector::next() {
// check that overall probability is 1
assert(fabs(cumulatedProbabilities[cumulatedProbabilities.size()-1]-1.0) < EPS);

// get a random number between 0 and 1
double number = genrand_real1();

// select next move
AbstractMove* pResult = NULL;
for(vector<double>::size_type i=0; i<cumulatedProbabilities.size();i++) {
  if(number<=cumulatedProbabilities[i]) {
    pResult=moves[i];
    break;
  }
}
assert(pResult != NULL);
return pResult;

}
```

print(move.getName(), "\t", move.getAcceptanceRate())	void MoveSelector::printRates() { cout << "--\n" "Move acceptance rates:\n"; for(vector<AbstractMove*>::size_type i=0; i<moves.size();i++) { cout << (*moves[i]).getName() << "\t"; cout << (*moves[i]).getAcceptanceRate() << endl; } }

class AbstractMove:
  def init(self, config, energy):
    self.config = config # Configuration object
    self.energyFunction = energy # Energy object
    self.counter = 0 # Counter for prepared moves
    self.accepted = 0 # Counter for accepted moves
    self.prepared = False # Flag indicating if a move is prepared
  def prepareMove(self):
    pass
 	
  def calcDeltaEnergy(self):
    # Implement calcDeltaEnergy logic here
    pass # Placeholder for the actual implementation
 	
  def performMove(self):
    pass
 	
  def reset(self):
    # Reset move state
    self.prepared = False # Set prepared flag to False
 	
  def getName(self):
    # Implement getName logic here
    pass # Placeholder for the actual implementation
 	
  def getAcceptanceRate(self):
    # Calculate acceptance rate
    if self.counter == 0:
      return 0.0 # Return 0 if no moves have been prepared
    else:
      return self.accepted / self.counter # Return acceptance rate as a float
	AbstractMove::AbstractMove(Configuration &config, const Energy &energy)
	:config(config), energyFunction(energy){
	counter=0;
	accepted=0;
	prepared=false;
	}
	
	AbstractMove::~AbstractMove() {
	}
	
	class AbstractMove { public: /** * @param config A Configuration, which is altered by the move * @param energy Energy object for the computation of delta energy */ AbstractMove(Configuration &config, const Energy &energy); virtual ~AbstractMove();
	/**
	 * Prepare and plan a move.
	 * @return boolean true - move is possible, false move is not possible
	 */
	virtual bool prepareMove() = 0;
	
	/**
	 * @return the energy that would result, if the move was performed
	 */
	virtual double calcDeltaEnergy() = 0;
	
	/**
	 * Performs a move. Move has to be prepared in advance.
	 */
	virtual void performMove() = 0;
	
	/**
	 * Resets a move. Attribute prepared will be false afterwards.

between 0 and 1	*/
	virtual void reset() { prepared = false; }
	
	/**
	 * @return name of the move
	 */
	virtual const char* getName() = 0;
	
	/**
	 * @return the acceptance rate of the move. Value between 0 and 1
	 */
	double getAcceptanceRate() { return double(accepted)/double(counter); }
	
	protected: /** * The Configuration that is altered by the move */ Configuration &config;
	/**
	 * Energy object for the computation of delta energy
	 */
	const Energy &energyFunction;
	
	/**
	 * Counts how many times the move was prepared
	 */
	long int counter;
	
	/**
	 * Counts how many times the move was accepted/performed
	 */
	long int accepted;
	
	/**
	 * State if the move was prepared
	 */
	bool prepared;
	
	};

<table>
<tr>
<td></td>
<td>

```
#endif /* ABSTRACTMOVE_H_ */
```

</td>
</tr>
<tr>
<td>

```
class AddMove(AbstractMove):
    def init(self, config, energy):
        super().init(config, energy)
        self.positions = config.getLength() - config.getNucLength() - 1 assert self.positions > 0
        self.nucStartPos = 0
    def prepareMove(self):
        self.prepared = False
        self.counter += 1

        assert self.config.getLength() > self.config.getNucLength()

        randomIndex = genrand_int32() % self.positions + 1  # +1 because 0 is not allowed
        if self.config.isStartPositionFree(randomIndex):
            self.nucStartPos = randomIndex
            del randomIndex
            self.prepared = True
        return self.prepared

    def calcDeltaEnergy(self):
        assert self.prepared == True
        center = self.nucStartPos + self.config.getNucLength() // 2
        return self.energyFunction.getAddEnergyDifference(center)

    def performMove(self):
        assert self.prepared == True
        self.accepted += 1
        self.config.addNucleosome(self.nucStartPos)
        self.prepared = False

    def __del__(self):
        pass  # Destructor doesn't contain any specific cleanup
```

</td>
<td>

```
AddMove::AddMove(Configuration &config, const Energy &energy)
: AbstractMove(config, energy){
// potential num of positions = length - nuc length - 1 (DNA margins)
positions   = config.getLength() - config.getNucLength() - 1;
assert( positions > 0);
nucStartPos = 0;
}

AddMove::~AddMove() {
}

bool AddMove::prepareMove() {
prepared = false;
counter++;
assert(config.getLength() > config.getNucLength());
// + 1 because 0 is not allowed
long int randomIndex = genrand_int32() % positions + 1;
if(config.isStartPositionFree(randomIndex) == true) {
nucStartPos = randomIndex;
prepared = true;
}
return prepared;
}

double AddMove::calcDeltaEnergy() {
assert(prepared == true);
long int center = nucStartPos+config.getNucLength()/2;
return energyFunction.getAddEnergyDifference(center);
}

void AddMove::performMove() {
assert(prepared == true);
accepted++;
```

</td>
</tr>
</table>

def getName(self):
      return "AddMove"
	config.addNucleosome(nucStartPos);
	prepared = false;
	}
	
	class AddMove : public AbstractMove { public: /** * @see AbstractMove */
	AddMove(Configuration &conf, const Energy &energy); virtual ~AddMove();
	virtual bool prepareMove();
	virtual double calcDeltaEnergy();
	virtual void performMove();
	virtual const char* getName() { return "AddMove"; };
	
	private: long int nucStartPos; long int positions; };
	#endif /* ADDMOVE_H_ */

class DeleteMove(AbstractMove):
   def init(self, config, energy):
      super().init(config, energy)
      self.nucIndex = 0 # num of potential positions with nucleosome coverage = length - 2 (DNA margins)
      self.positions = config.getLength() - 2 assert
      self.positions > 0
   def prepareMove(self):
      self.prepared = False
      self.counter += 1
 	
      # position 0 is not allowed, has to be DNA
      randomPos = genrand_int32() % self.positions + 1
      interval = self.config.getInterval(randomPos)
 	
      if interval.getType() == Interval.TYPE.NUC:
         self.prepared = True
         self.nucIndex = self.config.getNucIndex(randomPos)
         del randomPos
      return self.prepared
	DeleteMove::DeleteMove(Configuration &config, const Energy &energy)
	: AbstractMove(config, energy){
	nucIndex = 0;
	// num of potential positions with nucleosome coverage = length -2 (DNA margins)
	positions = config.getLength() - 2;
	assert(positions > 0);
	}
	
	DeleteMove::~DeleteMove() {
	}
	
	bool DeleteMove::prepareMove() {
	prepared = false;
	counter++;
	
	// position 0 is not allowed, has to be DNA
	long int randomPos = genrand_int32() % positions + 1;
	Interval interval = config.getInterval(randomPos);
	if(interval.getType() == Interval::NUC) {
	prepared = true;
	nucIndex = config.getNucIndex(randomPos);

 def calcDeltaEnergy(self):
      assert self.prepared == True
      center = self.config.getStartPositionOfNuc(self.nucIndex) + self.config.getNucLength() // 2
      return self.energyFunction.getAddEnergyDifference(center)
 	
   def performMove(self):
      assert self.prepared == True
      self.accepted += 1
      self.config.deleteNucleosome(self.nucIndex)
      self.prepared = False
   def getName(self):
      return "DeleteMove"
	}
	return prepared;
	}
	
	double DeleteMove::calcDeltaEnergy() {
	assert(prepared == true);
	long int center = config.getStartPositionOfNuc(nucIndex) + config.getNucLength()/2;
	return energyFunction.getDeleteEnergyDifference(center);
	}
	
	void DeleteMove::performMove() {
	assert(prepared == true);
	accepted++;
	config.deleteNucleosome(nucIndex);
	prepared = false;
	}

class ShiftMove(AbstractMove):
   def init(self, config, energy):
      super().init(config, energy)
      self.nucIndex = 0
      self.distance = 0
   def reset(self):
      super().reset()
      self.distance = 0
 	
   def prepareMove(self):
      self.prepared = False
      self.counter += 1
	ShiftMove::ShiftMove(Configuration &config, Energy &energy)
	: AbstractMove(config, energy){ nucIndex = 0; distance = 0; }
	ShiftMove::~ShiftMove() { }
	void ShiftMove::reset() { AbstractMove::reset(); distance = 0; }
	bool ShiftMove::prepareMove() { prepared = false; counter++;
	int num = config.getNumOfNucleosomes();
	if(num > 0) {
	   // select a random nucleosome
	   nucIndex = genrand_int32() % num;
	   distance = (genrand_int32() % (2*MAX_NUC_SHIFT)) - MAX_NUC_SHIFT;
	   // omit the zero
	   if (distance >= 0) {
	      distance += 1;

```
        num = self.config.getNumOfNucleosomes()
        if num > 0:
            # Select a random nucleosome
            self.nucIndex = genrand_int32() % num
            self.distance = (genrand_int32() % (2*MAX_NUC_SHIFT)) -
MAX_NUC_SHIFT
            # Omit the zero
            if self.distance == 0:
                self.distance += 1
            assert(self.distance != 0)
            assert -MAX_NUC_SHIFT <= self.distance <= MAX_NUC_SHIFT
            if self.config.canShiftNucleosome(self.nucIndex, self.distance) == True:
                self.prepared = True

        return self.prepared

    def calcDeltaEnergy(self):
        assert self.prepared
        nuc = self.config.getNucleosomeInterval(self.nucIndex)
        from_center_pos = nuc.getBegin() + self.config.getNucLength() // 2
        to_center_pos = nuc.getBegin() + self.distance + self.config.getNucLength()
// 2

        return self.energyFunction.getShiftEnergyDifference(from_center_pos,
to_center_pos)


    def performMove(self):
        assert self.prepared
        self.accepted += 1
        self.config.shiftNucleosome(self.nucIndex, self.distance)
        self.prepared = False
    def getName(self):
        return "ShiftMove"
```

```
    }
    assert(distance != 0);
    assert(-MAX_NUC_SHIFT <= distance && distance <= MAX_NUC_SHIFT);

    if ( config.canShiftNucleosome(nucIndex, distance) == true ) {
        prepared = true;
    }
}
return prepared;

}
double ShiftMove::calcDeltaEnergy() { assert(prepared == true);
Interval nuc = config.getNucleosomeInterval(nucIndex);

long int fromCenterPos = nuc.getBegin() + config.getNucLength()/2;
long int toCenterPos   = nuc.getBegin() + distance + config.getNucLength()/2;

return energyFunction.getShiftEnergyDifference(fromCenterPos, toCenterPos);

}
void ShiftMove::performMove() { assert(prepared == true); accepted++;
config.shiftNucleosome(nucIndex, distance); prepared = false; }
```

```
class PairShiftMove(AbstractMove):
    def init(self, config, energy):
        super().init(config, energy)
        self.nucIndex0 = 0
        self.nucIndex1 = 0
        self.distance = 0
    def reset(self):
        super().reset()
        self.distance = 0

    def prepareMove(self):
        self.prepared = False
        self.counter += 1

        num = self.config.getNumOfNucleosomes()
        if num >= 2:
            self.nucIndex0 = genrand_int32() % num
            self.nucIndex1 = (self.nucIndex0 + 1) % num

            self.distance = (genrand_int32() % (2 * MAX_NUC_PAIR_SHIFT)) -
MAX_NUC_PAIR_SHIFT
            if self.distance >= 0:
                self.distance += 1
            assert -MAX_NUC_PAIR_SHIFT <= self.distance <= MAX_NUC_PAIR_SHIFT

            if self.config.canShiftNucleosomePair(self.nucIndex0, self.nucIndex1,
self.distance):
                self.prepared = True

        return self.prepared

    def calcDeltaEnergy(self):
        assert self.prepared

        nuc0 = self.config.getNucleosomeInterval(self.nucIndex0)
```

```
PairShiftMove::PairShiftMove(Configuration &config, Energy &energy)
: AbstractMove(config, energy){ nucIndex0 = 0; nucIndex1 = 0; distance = 0; }
PairShiftMove::~PairShiftMove() { }
void PairShiftMove::reset() { AbstractMove::reset(); distance = 0; }
bool PairShiftMove::prepareMove() { prepared = false; counter++;
// one less because we shift pairs
int num   = config.getNumOfNucleosomes();
if(num >= 2) {
    // shift nucIndex and successor
    // select a random nucleosome
    nucIndex0 = genrand_int32() % num;
    // if nucIndex is the last nucleosome then
    // nucIndex+1 is again the first nucleosome
    nucIndex1 = (nucIndex0+1) % num;
    distance = (genrand_int32() % (2*MAX_NUC_PAIR_SHIFT))-MAX_NUC_PAIR_SHIFT;
    // omit the 0
    if (distance >= 0) {
        distance += 1;
    }
    assert(-MAX_NUC_PAIR_SHIFT <= distance && distance <= MAX_NUC_PAIR_SHIFT);

    if ( config.canShiftNucleosomePair(nucIndex0, nucIndex1, distance) == true ) {
        prepared = true;
    }
}
return prepared;

}
double PairShiftMove::calcDeltaEnergy() { assert(prepared == true);
Interval nuc0 = config.getNucleosomeInterval(nucIndex0);
Interval nuc1 = config.getNucleosomeInterval(nucIndex1);

long int fromCenterPos0 = nuc0.getBegin() + config.getNucLength()/2;
long int fromCenterPos1 = nuc1.getBegin() + config.getNucLength()/2;
long int toCenterPos0 = nuc0.getBegin() + distance + config.getNucLength()/2;
```

nuc1 = self.config.getNucleosomeInterval(self.nucIndex1)
 	
from_center_pos_0 = nuc0.getBegin() + self.config.getNucLength() // 2
 	
from_center_pos_1 = nuc1.getBegin() + self.config.getNucLength() // 2
 	
to_center_pos_0 = nuc0.getBegin() + self.distance + self.config.getNucLength() // 2
 	
to_center_pos_1 = nuc1.getBegin() + self.distance + self.config.getNucLength() // 2
 	
energy = self.energyFunction.getShiftEnergyDifference(from_center_pos_0, to_center_pos_0) + \
self.energyFunction.getShiftEnergyDifference(from_center_pos_1, to_center_pos_1)
 	
return energy
 	
def performMove(self):
assert self.prepared
self.accepted += 1
 	
if self.distance < 0:
self.config.shiftNucleosome(self.nucIndex0, self.distance)
self.config.shiftNucleosome(self.nucIndex1, self.distance)
elif self.distance > 0:
self.config.shiftNucleosome(self.nucIndex1, self.distance)
self.config.shiftNucleosome(self.nucIndex0, self.distance)
 	
self.prepared = False
 	
def getName(self):
	long int toCenterPos1 = nuc1.getBegin() + distance + config.getNucLength()/2;
	
	double energy = energyFunction.getShiftEnergyDifference(fromCenterPos0, toCenterPos0)
	+ energyFunction.getShiftEnergyDifference(fromCenterPos1, toCenterPos1);
	
	return energy;
	
	}
	void PairShiftMove::performMove() { assert(prepared == true); accepted++; // shift to the left if(distance < 0) { config.shiftNucleosome(nucIndex0, distance); config.shiftNucleosome(nucIndex1, distance); } else { // shift to right or neutral config.shiftNucleosome(nucIndex1, distance); config.shiftNucleosome(nucIndex0, distance); } prepared = false; }

return "PairShiftMove"	

class SimController:
    def init(self, config, energyFunction):
        self.config = config
        # initialize random number generator
        seed=27
        random.seed(seed)
 	
        print("Seed for random number generator:", seed)
 	
        self.moveSelector = MoveSelector()
        self.moveSelector.addMove(AddMove(config, energyFunction), ADD_RATE)
        self.moveSelector.addMove(DeleteMove(config, energyFunction), DELETE_RATE)
        self.moveSelector.addMove(ShiftMove(config, energyFunction), SHIFT_RATE)
        self.moveSelector.addMove(PairShiftMove(config, energyFunction), PAIR_SHIFT_RATE)
 	
        self.temperature = 0.0
 	
    def run(self, steps, stepsToSave, temperature):
        self.temperature = temperature
 	
        infoStepSize = self.computeInfoStepSize(steps)
 	
        for i in range(steps + 1):
            self.config.setTemperature(temperature)
            if i % infoStepSize == 0:
                print("\rProgress: {:.0f} % ".format(i / steps * 100), end="")
                sys.stdout.flush()
            self.step()
 	
        print("\rProgress: 100 % ")
	SimController::SimController(Configuration &config, Energy &energyFunction, ostream &simOut, ostream &energyOut) :config(config), simOut(simOut), energyOut(energyOut) {
	// initialize random number generator
	time_t seed;
	time(&seed);
	init_genrand((unsigned long) seed);
	
	cout << "Seed for random number generator: " << seed << endl;
	
	moveSelector.addMove(new AddMove(config, energyFunction), ADD_RATE);
	moveSelector.addMove(new DeleteMove(config, energyFunction), DELETE_RATE);
	moveSelector.addMove(new ShiftMove(config, energyFunction), SHIFT_RATE);
	moveSelector.addMove(new PairShiftMove(config, energyFunction), PAIR_SHIFT_RATE);
	
	temperature = 0.0;
	
	this->energyOut << "# Energy output over simulation run\n";
	this->energyOut << "# step energy" << endl;
	
	}
	SimController::~SimController() { }
	void SimController::run(long int steps, long int stepsToSave, double temperature) { this->temperature = temperature;
	long int infoStepSize = computeInfoStepSize(steps);
	
	for(long int i=0; i<=steps; i++) {
	
	    config.setTemperature(temperature);
	    if (i % stepsToSave == 0) {
	        writeEnergy(i);
	        writeConfig();
	    }
	    if (i % infoStepSize == 0) {

```
    print()
    self.moveSelector.printRates()
    print("Simulation completed")


def runAnnealing(self, steps, stepsToSave, startTemp, endTemp):
    assert startTemp > endTemp
    assert endTemp > 0.0
    assert steps > 0

    annealingSteps = 0

    if 0 < steps < 1000:
       annealingSteps = steps
    elif 1000 <= steps < 1000000:
       annealingSteps = steps // 10
    elif steps >= 1000000:
       annealingSteps = 100000
    else:
       raise Exception("Internal annealing step error")
```

```
      printf("\rProgress: %3.0f %% ",(double)i/(steps)*100 );
      fflush(stdout);
   }
   step();
}

printf("\rProgress: %3.0f %% ", 100.0);
cout << endl;
moveSelector.printRates();
cout << "Simulation completed" << endl;

}
void SimController::runAnnealing(long int steps, long int stepsToSave, double startTemp,
double endTemp) {
// checking the parameters
assert(startTemp > endTemp);
assert(endTemp > 0.0);
assert(steps > 0);

long int annealingSteps = 0;

if(0 < steps && steps < 1000)  {
   annealingSteps = steps;
} else if (1000 <= steps && steps < 1000000) {
   annealingSteps = steps/10;
} else if (1000000 <= steps) {
   annealingSteps = 100000;
} else {
   throw NucPosRunTimeException("Internal annealing step error",
         __FILE__, __LINE__);
}

cout << "Annealing steps: " << annealingSteps << endl;

assert(0 < annealingSteps && annealingSteps <= 100000);
```

```python
        print("Annealing steps:", annealingSteps)

        assert 0 < annealingSteps <= 100000

        annealingFactor = pow((endTemp / startTemp), (1.0 / annealingSteps))
        annealingStepSize = steps // annealingSteps
        self.temperature = startTemp
        self.config.setTemperature(self.temperature)

        infoStepSize = self.computeInfoStepSize(steps)

        for i in range(steps + 1):

            if i % infoStepSize == 0:
                print("\rProgress: {:.0f} %\tTemperature: {:.1f} K\t #Nucs: {:6}".format(i / steps * 100, self.temperature, self.config.getNumOfNucleosomes()), end="")
                sys.stdout.flush()

            self.step()

            if i % annealingStepSize == 0 and self.temperature > endTemp:
                self.temperature *= annealingFactor
                self.config.setTemperature(self.temperature)

        print("\rProgress: 100 %\tTemperature: {:.1f} K\t #Nucs: {:6}".format(self.temperature, self.config.getNumOfNucleosomes()))
        print()
        self.moveSelector.printRates()
        print("Simulation completed")
```

```cpp
// the simulation run

double annealingFactor = pow((endTemp/startTemp),(1.0/(double)annealingSteps) );
long int annealingStepSize = steps / annealingSteps;
temperature = startTemp;
config.setTemperature(temperature);

const long int infoStepSize = computeInfoStepSize(steps);

for(long int i=0; i<=steps; i++) {
    if (i % stepsToSave == 0) {
      writeEnergy(i);
      writeConfig();
    }

    if (i % infoStepSize == 0) {
      printf("\rProgress: %3.0f %%\tTemperature: %05.1f K\t #Nucs: %6li",
          (double)i/(steps)*100, temperature, config.getNumOfNucleosomes() );
      fflush(stdout);
    }

    step();

    if (i % annealingStepSize == 0 && temperature > endTemp) {
      temperature *= annealingFactor;
      config.setTemperature(temperature);
    }

}
printf("\rProgress: %3.0f %%\tTemperature: %05.1f K\t #Nucs: %6li",
     100.0, temperature, config.getNumOfNucleosomes() );
cout << endl;
moveSelector.printRates();
```

```python
def step(self):
    pMove = self.moveSelector.next_()

    success = pMove.prepareMove()
    self.config.increaseSteps()

    if success:
        deltaEnergy = pMove.calcDeltaEnergy()
        p = 1.0
        if deltaEnergy <= 0:
            p = 1.0
        else:
            p = math.exp(-deltaEnergy / (K_B * self.temperature))

        randomNumber = genrand_real1()
        if randomNumber <= p:
            pMove.performMove()
            self.config.addDeltaEnergy(deltaEnergy)
    else:
        pMove.reset()
```

```cpp
cout << "Simulation completed" << endl;

}
void SimController::step() {
        AbstractMove *pMove = moveSelector.next();

        bool success = pMove->prepareMove();
        config.increaseSteps();

        if (success){
            double deltaEnergy = pMove->calcDeltaEnergy();
            // Metropolis criteria
            double p = 1.0;
            if (deltaEnergy <= 0) {
                p = 1.0;
            } else {
                p = exp(-deltaEnergy/(K_B*temperature));
            }

            double randomNumber = genrand_real1();
            if (randomNumber <= p) {
                pMove->performMove();
                config.addDeltaEnergy(deltaEnergy);
            }
        } else {
            pMove->reset();
        }

}
long int SimController::computeInfoStepSize(long int steps) { long int infoStepSize = 1; const long int maxInfoSteps = 1000; if(steps / infoStepSize > maxInfoSteps) { infoStepSize = steps / maxInfoSteps; // add 1 in case of division rest if(steps % maxInfoSteps != 0) { infoStepSize += 1; } }
return infoStepSize;
```

def computeInfoStepSize(self, steps):
    infoStepSize = 1
    maxInfoSteps = 1000
    if steps / infoStepSize > maxInfoSteps:
        infoStepSize = steps // maxInfoSteps
        if steps % maxInfoSteps != 0:
            infoStepSize += 1
 	
    return infoStepSize
	}
	void SimController::writeEnergy(long int step) {
	    energyOut << step << "\t" << config.getEnergy() << endl;
	}
	void SimController::writeConfig() {
	    ConfigWriter::writeConfigAndSimInfo2Bed(config, simOut);
	}

def usage():
    print("\nUsage:\n\nNucPosSimulator <peak_output.tsv> <nucleosome_center_data.tsv.gz> <params.txt> [output-path]\n\n"
    "\t<peak_output.tsv> tsv input file with peaks generated by peak calling\n"
    "\t<nucleosome_center_data.tsv.gz> tsv.gz input file with nucleosome center data\n"
    "\t<params.txt> parameter file\n"
    "\t[output-path] path to an alternative output directory (optional)\n")
	void usage() {
	    cerr << "Usage:\n\nNucPosSimulator <reads.bed> <params.txt> [output-path]\n\n"
	        << "\t<reads.bed> BED input file with paired end reads\n"
	        << "\t<params.txt> parameter file\n"
	        << "\t[output-path] path to an alternative output directory (optional)\n";
	}

def main():
    try:
        if len(sys.argv) != 4 and len(sys.argv) != 5:
            usage()
            sys.exit(0)
        file_location = sys.argv[1]
peaks as additional input
        filename = sys.argv[2]
       parameterFilename = sys.argv[3]
 	
        outputFilebase = filename
	int main(int argc, char *argv[]) {
	
	    try {
	        if(argc != 3 && argc != 4) {
	            usage();
	            exit(0);
	        }
	
	        const char *filename = argv[1];
	        const char *parameterFilename = argv[2];
	
	        string outputFilebase = string(filename);

```python
    if len(sys.argv) == 5:
        outputDir = sys.argv[4]
        if os.path.isdir(outputDir):
            outputFilebase = os.path.join(outputDir, os.path.basename(filename))
        else:
            print("The output directory does not exist.")
```

```cpp
/*
 * Take output directory if specified
 */
if(argc == 4) {
    const string outputDir = string(argv[3]);
    // check if directory exists
    if ( Path::dirExists(outputDir) ) {
        stringstream pathStr;
        pathStr << outputDir;

        // Add a path separator in case it is missing
        if(outputDir[outputDir.length()-1] != PATH_SEP) {
            pathStr << PATH_SEP;
        }

        pathStr << Path::getBasename(string(filename));
        outputFilebase = pathStr.str();
    } else {
        stringstream errMsg;
        errMsg << "The output directory does not exist.";
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
    }
}

// get stream for simulation output
string simFilename = outputFilebase + ".sim";
cerr << simFilename;
ofstream simOut;
simOut.open(simFilename.c_str());
if( simOut == NULL) {
    stringstream errMsg;
    errMsg << "Unable to write file " << simFilename;
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
}
```

```
	simOut << "# Snapshots of a simulation run. Every block \n"
		<< "# contains a snapshot of the nucleosome configuration.\n"
		<< endl;

	// get stream for energy output
	string energyFilename = outputFilebase + ".energyOut";
	ofstream energyOut;
	energyOut.open(energyFilename.c_str());
	if( energyOut == NULL) {
	  stringstream errMsg;
	  errMsg << "Unable to write file " << energyFilename;
	  throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
	}

	// get stream for peak output
	string distFilename = outputFilebase + ".occupancy";
	ofstream distOut;
	distOut.open(distFilename.c_str());
	if( distOut == NULL) {
	  stringstream errMsg;
	  errMsg << "Unable to write file " << distFilename;
	  throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
	}

#ifdef VERBOSE
	// get stream for peak output
	string peaksFilename = outputFilebase + ".peaks";
	ofstream peaksOut;
	peaksOut.open(peaksFilename.c_str());
	if( peaksOut == NULL) {
	  stringstream errMsg;
	  errMsg << "Unable to write file " << peaksFilename;
	  throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
	}
```

```
    simSteps = None
    stepsToSave = None
    nucLength = None
    sigma = None
    annealing = None
    startTemperature = None
    endTemperature = None
    bindingEnergy = None
    temperature = None

    print("\n-----------------------------------------")
    readReader = ReadReader(filename)
    locusBegin = readReader.getLocusBegin()
    locusEnd = readReader.getLocusEnd()
    chrom = readReader.getChromosome()
    length = locusEnd - locusBegin
    minVal = readReader.getMin()
```

```
    // get stream for energy function output
    string energyFuncFilename = outputFilebase + ".energyFunc";
    ofstream energyFuncOut;
    energyFuncOut.open(energyFuncFilename.c_str());
    if( energyFuncOut == NULL) {
      stringstream errMsg;
      errMsg << "Unable to write file " << energyFuncFilename;
      throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
    }

#endif

    /*********************************************************************
     * Begin parameter processing
     *********************************************************************/
    // read paramters from file
    ParameterList paramList =
ParameterReader::getParametersFromFile(parameterFilename);

    /*
     * SimSteps
     */
    long int steps = paramList.getValue("SimSteps");

    if (steps <= 0) {
      stringstream errMsg;
      errMsg << "Please check parameter file. SimSteps must be greater than 0.";
      throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
    }

    /**
     * StepsToSave
     */
    long int stepsToSave = paramList.getValue("StepsToSave");
```

```python
    maxVal = readReader.getMax()

    with open(parameterFilename, "r") as paramFile:
        for line in paramFile:
            key, value = line.strip().split("\t")
            if key == "SimSteps":
                if value == 'default':
                    simSteps = 5*length
                else:
                    simSteps = int(value)
                del length
            elif key == "NucLength":
                nucLength = int(value)
            elif key == "SmoothingSigma":
                sigma = float(value)
            elif key == "Annealing":
                annealing = bool(int(value))
            elif key == "StartTemperature":
                startTemperature = float(value)
            elif key == "EndTemperature":
                endTemperature = float(value)
            elif key == "BindingEnergy":
                bindingEnergy = float(value)
            elif key == "Temperature":
                temperature = float(value)

    if None in (simSteps, stepsToSave, nucLength, sigma, annealing,
bindingEnergy):
        print("Missing or invalid parameter value.")
```

```cpp
if (steps % stepsToSave != 0) {
    stringstream errMsg;
    errMsg << "Please check parameter file. "
           << "SimSteps have to be a multiple of StepsToSave.";
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
}

/*
 * NucLength
 */
long int nucLength = (long int) paramList.getValue("NucLength");

if (nucLength <= 0) {
    stringstream errMsg;
    errMsg << "Please check parameter file. NucLength must be greater than 0.";
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
}

/**
 * Smoothing sigma
 */
double sigma = paramList.getValue("SmoothingSigma");

if (sigma <= 0) {
            stringstream errMsg;
            errMsg << "Please check parameter file. SmoothingSigma must"
                  << " be greater than 0.";
            throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
}

/*
 * Annealing
 */
if (paramList.getValue("Annealing") != 0 &&
        paramList.getValue("Annealing") != 1.0) {
```

```
  stringstream errMsg;
  errMsg << "Please specify parameter 'Annealing' with 0 (false) or"
      << " 1 (true) in parameter file";
  throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
}

bool annealing = paramList.getValue("Annealing");

double startTemperature = 0.0;
double endTemperature = 0.0;
if (annealing) {
  /*
   * StartTemperature and EndTemperature
   */
  startTemperature = paramList.getValue("StartTemperature");
  endTemperature = paramList.getValue("EndTemperature");

  // checking the parameters
  if (startTemperature <= endTemperature) {
    stringstream errMsg;
    errMsg << "Please check parameter file. StartTemperature has to be greater than "
        << "EndTemperature in Simulated Annealing run.";
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
  }

  if (endTemperature <= 0) {
    stringstream errMsg;
    errMsg << "Please check parameter file. EndTemperature has to be greater than 0 K.";
    throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
  }
}

/**
 * BindingEnergy
```

```
  print("\n------------------------------------------")

     print("Simulated Annealing : ", annealing)

     name = os.path.basename(filename)
     parent_dir = os.path.dirname(filename)

     pEnergy = EnergyFactory(parent_dir,name,readReader.getReads(),
locusBegin, locusEnd).give_energy(sigma, bindingEnergy)
Compressed into one line to minimize memory allocation

     del readReader
     gc.collect()
     print("Binding energy:", pEnergy.get_binding_energy())

     if annealing:
        print("Start temperature:", startTemperature)
        print("End temperature:", endTemperature)
```

```
      */
     double bindingEnergy = paramList.getValue("BindingEnergy");

     /*****************************************************************
      * End parameter processing
      ****************************************************************/

     cout << "\n----------------------------------------\n";

     ReadReader readReader(filename);

     vector< pair<long int, long int> >* pReads = readReader.getReads();
     const long int locusBegin = readReader.getLocusBegin();
     const long int locusEnd   = readReader.getLocusEnd();
     const string chrom = readReader.getChromosome();

     cout << "Simulated Annealing :  " << annealing << "\n";

     EnergyFactory ef(pReads, locusBegin, locusEnd);
     Energy* pEnergy = ef.giveEnergy(sigma, bindingEnergy);
     delete pReads;

     cout << "Binding energy: " << pEnergy->getBindingEnergy() << endl;

     ef.printProbabilities(distOut);

#ifdef VERBOSE
     ef.printFrequencies(peaksOut);
     pEnergy->printValues(energyFuncOut);

     peaksOut.close();
     energyFuncOut.close();
#endif
     Configuration config(locusBegin,locusEnd-locusBegin, nucLength, chrom);
```

```
        start_nucs = process_file(file_location)

        config = Configuration(name,start_nucs,minVal, maxVal - minVal,locusBegin,locusEnd-locusBegin, nucLength, chrom)
        del start_nucs

        SimController(config, pEnergy).runAnnealing(simSteps, stepsToSave, startTemperature, endTemperature)
        pEnergy.cleanup()
        del pEnergy
        gc.collect()






    bedFilename = outputFilebase + ".result.bed"
        with open(bedFilename, "w") as bedOut:
            config_writer = ConfigWriter()  # Create an instance of ConfigWriter
            config_writer.writeConfig2Bed(config, bedOut)  # Call the method on the instance


    except NucPosIOException as e:
        print("\nERROR - IO exception:")
        print(e.getMessage(), "\n")
        return -1
    except NucPosRunTimeException as e:
        print("\nERROR - Runtime exception:")
        print(e.getMessage(), "\n")
        return -1
```

```
SimController simcontroller(config, *pEnergy, simOut, energyOut);

if (annealing) {
    cout << "Start temperature: " << startTemperature << endl;
    cout << "End temperature: " << endTemperature << endl;
    simcontroller.runAnnealing(steps,
            stepsToSave,
            startTemperature,
            endTemperature);

    // get stream for output of last config
    string bedFilename = outputFilebase + ".result.bed";
    ofstream bedOut;
    bedOut.open(bedFilename.c_str());
    if( bedOut == NULL) {
        stringstream errMsg;
        errMsg << "Unable to write file " << bedFilename;
        throw NucPosIOException(errMsg.str(), __FILE__, __LINE__ );
    }

    // print last config as bed file
    ConfigWriter::writeConfig2Bed(config, bedOut);
    bedOut.close();
} else {
    double temperature = REFERENCE_TEMPERATURE;
    if (paramList.keyExists("Temperature")) {
        temperature = paramList.getValue("Temperature");
    }
    cout << "Temperature: " << temperature << endl;
    simcontroller.run(steps, stepsToSave, temperature);
}

delete pEnergy;

distOut.close();
```

if name == "main": main()	energyOut.close();
	    simOut.close();
	
	  } catch(NucPosIOException &e) {
	    cerr << "\nERROR - IO exception:\n";
	    cerr << e.getMessage() << "\n" << endl;
	    return -1;
	  } catch(NucPosRunTimeException &e) {
	    cerr << "\nERROR - Runtime exception:\n";
	    cerr << e.getMessage() << "\n" << endl;
	    return -1;
	  }
	  return 1;
	}

	ParameterReader::ParameterReader() { }
	ParameterReader::~ParameterReader() { }
	ParameterList ParameterReader::getParametersFromFile(const char *filename) {
	ifstream inputFile (filename);
	if (inputFile.is_open() == false) {
	  stringstream errMsg;
	  errMsg << "Unable to open input file: \"" << filename << "\"\n";
	  throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
	}
	
	const int max_length = 100;
	char line[max_length] = {};
	char key[max_length] = {};
	
	double value;
	
	ParameterList paramList;
	
	while (inputFile.good()) {
	  inputFile.getline(line, max_length);

```
  // sscanf returns the numbers of successful reads
  if (sscanf(line, "%99s%lf", key, &value) == 2) {
    // key duplication is not allowed
    if (paramList.keyExists(key) == true) {
      stringstream errMsg;
      errMsg << "Duplication of key: " << key
          << "\nin parameter file: " << filename;
      throw NucPosIOException(errMsg.str(), __FILE__, __LINE__);
    }
    assert(paramList.keyExists(key) == false);
    paramList.addKeyValue(key, value);
  }
}

inputFile.close();

return paramList;

}
class ParameterReader { public: ParameterReader(); virtual ~ParameterReader();
/**
 * Reads parameters from a file (key value format)
 */
static ParameterList getParametersFromFile(const char *filename);

};
#endif /* PARAMETERREADER_H_ */
```

```
string Path::getBasename(string path) {
  size_t i = path.rfind (PATH_SEP, path.length());
  string result = path;
  if (i != string::npos) {
    result = path.substr(i+1, path.length()-i);
  }
```

```
    return result;
}
bool Path::dirExists(const string path) {
    struct stat status;
    // stat returns 0 if the operation was successful
    int check = stat( path.c_str(), &status );
    bool result = false;
    if (check == 0) {
        // check if directory exists
        if ( status.st_mode & S_IFDIR ) {
            result = true;
        }
    }
    return result;
}
class Path {
public:
    Path() {;}
    virtual ~Path() {;}
    /**
     *  Gets the filename out of a path
     *  @return the filename
     */
    static string getBasename(const string path);

    /**
     * Checks if the path is an valid directory
     */
    static bool dirExists(const string path);
};
#endif /* PATH_H_ */
```