VitBerBioFysikk

February 5, 2024

1 TMA4320 Biophysics project

1.0.1 Oliver Ruden, Åsmund Mjøs & Astrid Mysterud

Polymers are the building blocks of DNA, RNA and proteins. Polymers themselves consist of repeating units called monomers. In this project, polymers will be represented numerically in order to make energy calculations as the polymers rotate around their monomers. By using the Monte Carlo method, as described in the project description, we achieve rotations that account for realistic energy changes. The first part of the project mainly consists of creating functions needed for rudimentary polymer-functionality. The focus of the second part of the project is to investigate how rotations of the polymer and its energy varies under different conditions, such as temperature.

In this project, a polymer with N monomers will be represented by a $N \times 2$-array containing the N sets of x- and y-coordinates of their monomers. These x- and y-coordinates will all be bounded by $0 \leq x, y \leq N + 1$. We always rotate with the middle held constant, to ensure the inequalities are upheld. Furthermore, the reason $N + 1$ is the bound is that in the case of the polymer being of even length, the middle isn't uniquely defined, and we have chosen to take the middle rounded up. Thus, if the entire polymer is rotated 180°, it will need the grid to be of length $N + 1$. This will also always be sufficient, as the longest tail is of length $N/2$ rounded up, and the middle plus this length is less than or equal to $N + 1$.

The following function `createPolymer`, as described in task **1b)**, takes the argument N and returns a polymer of size N shaped as a horizontal line. To determine the y-coordinate, integer division is used.

[ ]:
```
# First, we import necessary libaries
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
import timeit
import time
plt.rcParams.update({'font.size': 20}) # change font-size of the plot

"""
Get time taken for the entire code
"""
startTime = time.time()
```

```
[ ]: def createPolymer(N):
         """
         Create a straight polymer.
         Input:
         N: Length of the polymer
         """
         polymer = np.zeros((N, 2))
         """
         Put the monomers at a straight line at y = N / 2 rounded down
         """
         polymer[:, 1] = N // 2
         polymer[:, 0] = np.array([x for x in range(N)])
         return polymer
```

In order to follow the rules that determine valid polymers, the energy will remain unchanged after a rotation around an endpoint. The energy of a polymer is determined by the distance between its monomers. When a polymer is rotated around an endpoint, the intermonomer distance is constant for all pairs of monomers, which implies that they are neighbours iff they were neighbours, and as a result the energy of the polymer does not change. Rotating around the endpoints is also a physically meaningless rotation, since it does not change the structure of the polymer, it just changes the direction of which the polymer lies in the two-dimensional grid.

In algorithm 1 and 2 from the project description, we are to choose a random monomer to rotate around. As explained in the latter paragraph, the physical interpretation of rotating around the endpoints, as far as we know, is just changing the direction of which the polymer lies in the grid. Therefore, excluding rotation around endpoints would give a better understanding of polymer behavior.

Inclusion and exclusion of edge monomers result in slighty different statistics, for example the percentage of legal rotations or the time it takes for an energy jump. This difference in statistics is negligible for larger polymers. However, for polymers of few monomers (e.g. under 10), this difference is significant. Since we have decided to exclude rotation around endpoints, it is important to be aware of this statistical difference.

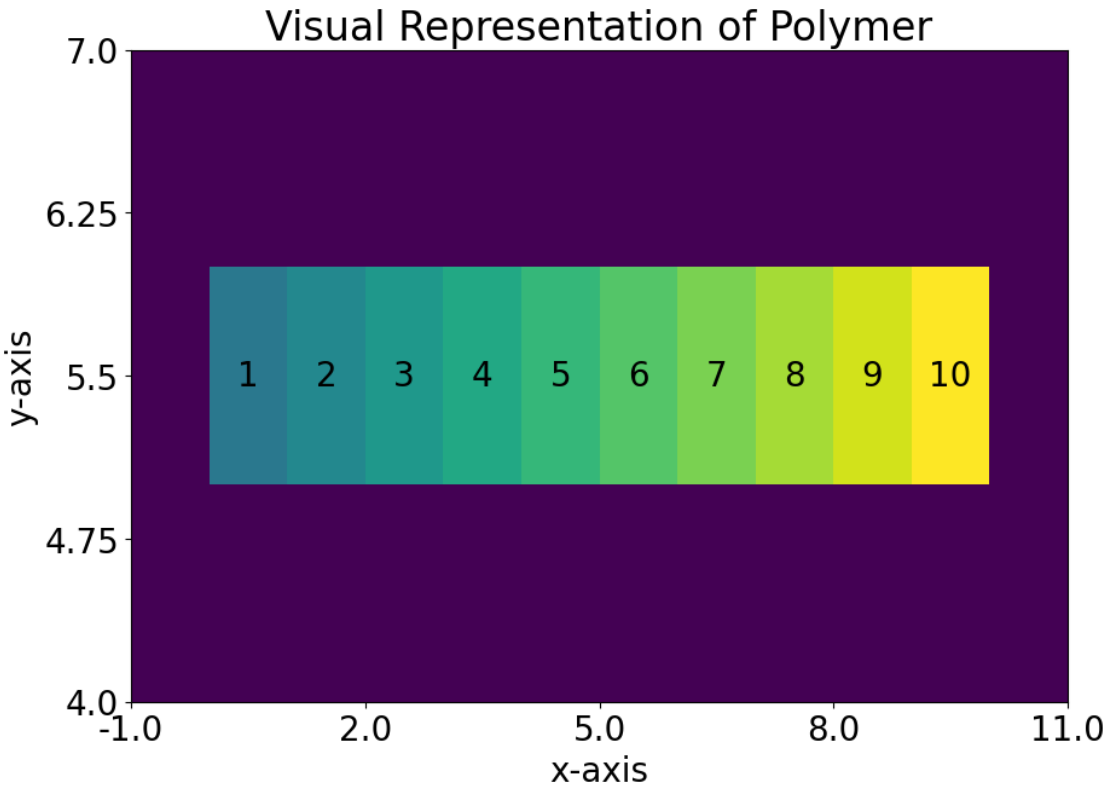
The following code implements a function that visualizes the polymer, as asked for in task **1d)**. The polymer is shown in yellow/green squares showing its monomer number. To zoom in where the polymer is situated in the $(N+1) \times (N+1)$-grid, we illustrate the polymer in a grid that ranges from the minimum x-value to the maximum x-value held by a monomer, likewise in the y-direction.

We change the background value of the grid to $-\lfloor N/2 \rfloor$, to better distinguish the polymer from the background, but also to still be able to uniquely identify monomers. Monomers with small indices are represented by darker blue/green, while monomers with greater indices are represented by brighter yellow.

```
[ ]: def illustratePolymer(polymer):
         """
         Illustrate a polymer in a 2-d grid where all non-occupied spaces have the␣
      ↪same color and the spaces monomers occupy have a color and number
         corresponding to their monomer number
```

```
    Input:
    polymer: The polymer to illustrate
    """
    N = len(polymer)
    """
    Find the x- and y-bounds of the grid
    """
    minx, maxx = np.min(polymer[:,0]), np.max(polymer[:,0])
    miny, maxy = np.min(polymer[:,1]), np.max(polymer[:,1])
    """
    Initialize the grid with -N//2 as background value to highlight the polymer
    """
    grid = np.zeros((int(maxy-miny)+3, int(maxx-minx)+3))
    grid -= N//2

    """
    Customizing the plot
    """
    plt.figure(figsize = (10,7))
    plt.title("Visual Representation of Polymer")
    plt.xlabel(r"x-axis")
    plt.ylabel(r"y-axis")

    """
    Loop through all monomers of the input polymer and place their monomer␣
↪number at the correct spot in the grid, both in term of text and value
    """
    for monomerNumber in range(N):
        x = int(polymer[monomerNumber, 0]-minx+1)
        y = int(polymer[monomerNumber, 1]-miny+1)
        grid[y, x] = monomerNumber + 1
        plt.text(x+0.5,y+0.5,monomerNumber+1, ha = 'center', va = 'center')

    """
    Customize the axis so the correct coordinates are put
    """
    plt.xticks(ticks = np.arange(0,int(maxx-minx+3)+int(maxx-minx+3)/
↪4,int(maxx-minx+3)/4),labels = np.
↪arange(minx-1,minx-1+int(maxx-minx+3)+int(maxx-minx+3)/4,int(maxx-minx+3)/4))
    plt.yticks(ticks = np.arange(0,int(maxy-miny+3)+int(maxy-miny+3)/
↪4,int(maxy-miny+3)/4),labels = np.
↪arange(miny-1,int(maxy-miny+3)+int(maxy-miny+3)/4+miny-1,int(maxy-miny+3)/4))
    plt.pcolormesh(grid)
    plt.show()

"""
Here, we visualize the horizontal polymer from task 1b) with N=10
```

```
"""
illustratePolymer(createPolymer(10))
```



While rotating a polymer around its monomers, one easily ends up with a polymer that does not meet the requirements as listed in task **1e)**. The following function receives a polymer and its size N, and checks whether it is valid or not. The function `validPolymer` returns `True` if the polymer is valid, and `False` otherwise. To make sure that no monomers have the same coordinates, we initialize the set `coordinateSet`, where we add the monomer-coordinates as we loop through the monomers of the polymer. Once a monomer with already taken coordinates is encountered, `validPolymer` returns `False`. If a monomer has unique coordinates (x, y), the tuple (x, y) is added to `coordinateSet`. To check that $m-1$ and $m+1$ are the closest neighbours of monomer m, we find the x- and y-distance to the monomer preceding m. If the sum of the x- and y-distance is not 1, the preceding monomer cannot be one of the closest neighbours of m.

To decrease runtime, `validPolymer` is compiled by using `@jit` from the `numba`-library. We can use `@jit` here, as it doesn't call on any other functions making it easy to apply it. Furthermore, it uses a for loop, one of the things that `@jit` is able to speed up drastically. As `validPolymer` is a frequently called upon function later in the code, this speed-up will be quite beneficial.

[ ]:
```
@jit
def validPolymer(polymer, N):
    """
```

```
    Check if a polymer is valid.
    Input:
    polymer: The polymer to check.
    N: The supposed length of the polymer.

    Check if the polymer has N monomers
    """
    if len(polymer) != N:
        return False

    """
    Check if coordinates are unique, uses set and tuples for an average of␣
↪constant time to check if an element in contained
    """
    coordinateSet = set()
    coordinateSet.add((polymer[0, 0], polymer[0, 1]))

    for monomerNumber in range(1, N):

        if (polymer[monomerNumber, 0], polymer[monomerNumber, 1]) in␣
↪coordinateSet:
            return False
        else:
            coordinateSet.add((polymer[monomerNumber, 0],␣
↪polymer[monomerNumber, 1]))

        """
        Check taxicab-distance, should be 1 for neighbours.
        """
        xDiff = np.abs(polymer[monomerNumber, 0] - polymer[monomerNumber - 1,␣
↪0])
        yDiff = np.abs(polymer[monomerNumber, 1] - polymer[monomerNumber - 1,␣
↪1])
        if xDiff + yDiff != 1:
            return False

    return True
```

Next, we are to implement rotation for the polymer. Task **1f)** states that the function should have the following input parameters: a polymer, a coordinate of a monomer, and the direction of rotation. In algorithm 1 and 2, which use the function that rotates the polymer, a random monomer is chosen to rotate around. We find it faster and much simpler to find a random monomer number to rotate around, rather than a random coordinate to rotate around. If we were to follow the prescribed input in task **1f)**, we would randomly generate a monomer number, find the coordinates corresponding to that monomer number and input that into `rotatePolymer`, here, we would convert it back to monomer number before continuing as the monomer number is necessary for our implementation of the rotation algorithm. Thus, we find it easier and more efficient

to implement `rotatePolymer` with monomer number as input. If `rotatePolymer` had monomer coordinate as input, the monomer number could be collected as shown in the following function `coordToMonomerNumber`. `coordToMonomerNumber` receives an input polymer and coordinate, and returns monomer number. This way we could have just had the monomer coordinate as input in `rotatePolymer` and found monomer number from `coordToMonomerNumber`.

[ ]:
```
def coordToMonomerNumber(polymer, coord):
    """
    To convert a coordinate in a polymer to a monomer number
    Input:
    polymer: The polymer which containes the coordinate
    coord: The coordinate which to find the corresponding monomer number
    """
    for monomerNumber in range(len(polymer)):
        if polymer[monomerNumber].all() == np.array(coord).all():
            return monomerNumber + 1
    return None
```

Next code block implements the rotating function `rotatePolymer`. The function `rotatePolymer` receives an input polymer and a monomer number which the polymer will rotate around. `rotatePolymer` also receives a boolean `positiveDirection`. `positiveDirection=True` rotates the polymer in the positive direction, while `positiveDirection=False` rotates the polymer in the negative direction.

The placement of each monomer in the polymer after a rotation around a monomer m_{rot}, is determined by it's relative distance in x-direction Δx to m_{rot} and it's relative distance in y-direction Δy to m_{rot}. We look at the monomers positions relative to m_{rot} as free vectors, and since we will either be rotating them in positive or negative direction, we can simply multiply by the rotation matrix, either positive direction, or negative direction.

In positive rotation, the original vector $(1, 0)$ will go to $(0, 1)$ and the vector $(0, 1)$ will go to $(-1, 0)$. In negative rotation, the original vector $(1, 0)$ will go to $(0, -1)$ and $(0, 1)$ will go to $(1, 0)$ in negative rotation. This is essentially all the information we need to implement rotation, since rotation is a linear operation, and these two vectors forms an orthonormal basis for $\mathbb{R}^2$.

To decrease runtime, `rotatePolymer` is compiled by using `@jit` from the `numba`-library.

[ ]:
```
@jit
def rotatePolymer(polymer, monomer, positiveDirection):
    """
    Rotates a polymer at a given monomer in the direction specified.
    Input:
    polymer: The polymer to rotate
    monomer: The monomer to rotate the polymer around (1-indexed, as we decided␣
 ↪previously)
    positiveDirection: It is 1 if we are supposed to rotate in the positive␣
 ↪direction, and 0 for rotation in the negative direction.
```

```
    Setting up for rotation by finding the index of the monomer and its x and y coordinates, alongside finding the index of the middle and
    initializing a new polymer array, as working inplace makes undoing the rotation if it is invalid more difficult.
    """
    monomer -= 1
    middleMonomer = len(polymer) // 2
    x, y = polymer[monomer]
    newPolymer = np.zeros((len(polymer),2))

    if middleMonomer > monomer:
        """
        If the monomer is on the left half of the polymer array, the right side of the monomer is copied over, while the left is rotated using
        the rotation matrix.
        """
        newPolymer[monomer:] = polymer[monomer:]
        newPolymer[:monomer, 0] = (2 * positiveDirection - 1) * (polymer[:monomer, 1] - y) + x
        newPolymer[:monomer, 1] = (1 - 2 * positiveDirection) * (polymer[:monomer, 0] - x) + y
        """
        Positive direction:
        delta x = delta y
        delta y = - delta x
        Negative direction:
        delta x = - delta y
        delta y = delta x

        Also uses that True is represented as 1 and False as 0.
        """
        return newPolymer
    """
    If the monomer is on the right half of the polymer array, the left side of the monomer is copied over, while the right is rotated using
    the rotation matrix.
    """
    newPolymer[:monomer+1] = polymer[:monomer+1]
    newPolymer[monomer+1:,0] = (1-2*positiveDirection)*(polymer[monomer+1:,1]-y)+x
    newPolymer[monomer+1:,1] = (2*positiveDirection-1)*(polymer[monomer+1:,0]-x)+y
    """
        Positive direction:
        delta y = delta x
        delta x = - delta y
```

```
        Negative direction:
        delta y = - delta x
        delta x = delta y
    """
    return newPolymer
```

Now that a polymer is able to rotate, we implement functionality to rotate multiple times as described in task **1g)**. `rotateNTimes` implements algorithm 1 from the project description and takes in the polymer size N and the number of wanted rotations N_s.

[ ]:
```
def rotateNTimes(N, Ns):
    """
    Attempt to rotate a polymer of length N Ns times.
    Input:
    N: The length of the polymer to rotate
    Ns: The number of rotation to attempt
    """
    rotationsMade = 0
    polymer = createPolymer(N)

    """
    Attempting to use rotatePolymer Ns times,
    checking if the rotated polymer is valid or not
    """
    for i in range(Ns):
        monomer = np.random.randint(2, N)
        positivRetning = np.random.randint(0, 2)

        twistedPolymer = rotatePolymer(polymer, monomer, positivRetning)
        if validPolymer(twistedPolymer, N):
            rotationsMade += 1
            polymer = twistedPolymer

    """
    Returning rotated polymer and the number of valid rotations
    """
    return polymer, rotationsMade
```

To illustrate and compare the end results of the two polymers in a appropriate way, we decided to implement the plotting function from task **2c)** here, so that we can illustrate the plot next to each other. The logic is quite similar to the logic in `illustratePolymer`, just that we do it in two functions, to avoid a lot of duplicate work.

[ ]:
```
def illustrationOfOnePolymer(polymer):
    """
    Input:
    polymer: The polymer to create a grid for
    """
```

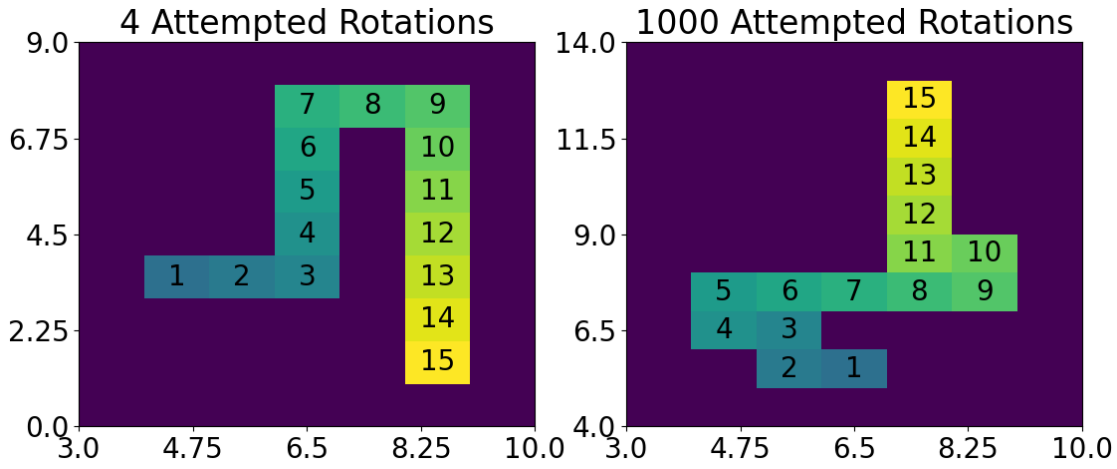
```
    N = len(polymer)                                    # calculate the size of the␣
↪input polymer
    """
    Getting the dimensions of the needed grid by finding the min and max value␣
↪for both the x and y axis.
    """
    minx, maxx = np.min(polymer[:,0]), np.max(polymer[:,0])
    miny, maxy = np.min(polymer[:,1]), np.max(polymer[:,1])
    """
    Creating the grid and placing the monomers in the middle, with the␣
↪non-occupied spaces having a value of N/2 rounded down and the monomers
    having the value of their monomerNumber (1,...,N)
    """
    grid = np.zeros((int(maxy-miny)+3, int(maxx-minx)+3))
    grid -= N//2
    for monomerNumber in range(N):                      # loop through all monomers␣
↪of the input polymer
        x = int(polymer[monomerNumber, 0]-minx+1)           # collect␣
↪x-coordinate of the monomer relative to the grid
        y = int(polymer[monomerNumber, 1]-miny+1)           # collect␣
↪y-coordinate of the monomer relative to the grid
        grid[y, x] = monomerNumber + 1
    return grid,minx,maxx,miny,maxy



def multiplePlotsPolymers(polymer1,polymer2, title1,title2):
    """
    Input:
    polymer1, polymer2: The two polymers to be illustrated
    title1, title2: The title over the respective plots of the polymers
    """
    fig, (ax1,ax2) = plt.subplots(nrows = 1, ncols = 2, figsize = (13,5))␣
↪#Initialising two plots next to each other

    #Sublot 1
    """
    Getting the correct grid and plotting the data in a pcolormesh for the␣
↪first polymer
    """
    N = len(polymer1)
    grid_1,minx,maxx,miny,maxy = illustrationOfOnePolymer(polymer1)
    ax1.pcolormesh(grid_1)
    ax1.set(title = title1)
    """
```

```
    Placing the monomer numbers over the monomers in the plot for the first␣
↪polymer.
    """
    for monomerNumber in range(N):
        x = int(polymer1[monomerNumber, 0]-minx+1)
        y = int(polymer1[monomerNumber, 1]-miny+1)
        ax1.text(x+0.5,y+0.5,monomerNumber+1, ha = 'center', va = 'center')
    """
    Customising the axis for the first polymer, so that the correct x and y␣
↪coordinates are illustrated
    """
    ax1.set_xticks(ticks = np.arange(0,int(maxx-minx+3)+int(maxx-minx+3)/
↪4,int(maxx-minx+3)/4),labels = np.
↪arange(minx-1,minx-1+int(maxx-minx+3)+int(maxx-minx+3)/4,int(maxx-minx+3)/4))
    ax1.set_yticks(ticks = np.arange(0,int(maxy-miny+3)+int(maxy-miny+3)/
↪4,int(maxy-miny+3)/4),labels = np.
↪arange(miny-1,int(maxy-miny+3)+int(maxy-miny+3)/4+miny-1,int(maxy-miny+3)/4))
    #Subplot 2
    """
    Getting the correct grid and plotting the data in a pcolormesh for the␣
↪first polymer
    """
    N = len(polymer2)
    grid_2,minx,maxx,miny,maxy = illustrationOfOnePolymer(polymer2)
    ax2.pcolormesh(grid_2)
    ax2.set(title = title2)
    """
    Placing the monomer numbers over the monomers in the plot for the second␣
↪polymer.
    """
    for monomerNumber in range(N):
        x = int(polymer2[monomerNumber, 0]-minx+1)
        y = int(polymer2[monomerNumber, 1]-miny+1)
        ax2.text(x+0.5,y+0.5,monomerNumber+1, ha = 'center', va = 'center')
    """
    Customising the axis for the second polymer, so the correct x and y␣
↪coordinates are illustrated
    """
    ax2.set_xticks(ticks = np.arange(0,int(maxx-minx+3)+int(maxx-minx+3)/
↪4,int(maxx-minx+3)/4),labels = np.
↪arange(minx-1,minx-1+int(maxx-minx+3)+int(maxx-minx+3)/4,int(maxx-minx+3)/4))
    ax2.set_yticks(ticks = np.arange(0,int(maxy-miny+3)+int(maxy-miny+3)/
↪4,int(maxy-miny+3)/4),labels = np.
↪arange(miny-1,int(maxy-miny+3)+int(maxy-miny+3)/4+miny-1,int(maxy-miny+3)/4))
    """
    Plot the two colormeshes
```

```
    """
    plt.show()
```

[ ]:
```
"""
Ns=4:
"""

polymerN4, rotationsN4 = rotateNTimes(15, 4)
print(f"A polymer of length N=15 trying to rotate Ns=4 times, had {rotationsN4}␣
 ↪successful rotations.")
np.savetxt('polymerArray15_4.txt', polymerN4)
"""
Ns=1000:
"""

polymerN1000, rotationsN1000 = rotateNTimes(15,1000)
print(f"A polymer of length N=15 trying to rotate Ns=1000 times, had␣
 ↪{rotationsN1000} successful rotations.")
np.savetxt('polymerArray15_1000.txt', polymerN1000)

multiplePlotsPolymers(polymerN4, polymerN1000, "4 Attempted Rotations", "1000␣
 ↪Attempted Rotations")
```

```
A polymer of length N=15 trying to rotate Ns=4 times, had 3 successful
rotations.
A polymer of length N=15 trying to rotate Ns=1000 times, had 561 successful
rotations.
```



One of the main problems within the computation time as of now is the time it takes to rotate the polymer. With the implementation we have used up to now, we split the cases so that we always only rotate the shorter side of the polymer. Thus on average, the length of the chain we rotate will be $N/2 \cdot 1/2 = N/4$. As we change both the x- and the y-value for each of these monomers every

time, we alter $N/2$ numbers on average per rotation. To lower this number, we reimplement the structure of the polymer. Instead of having a $N \times 2$ array which tracks the polymers coordinates, we use the fact that the polymer must be connected.

Thus, we create our own object called `Middle`, which tracks the x- and y-coordinates of the middle monomer and contains two arrays. The first array, called `beforeMiddle`, is of length the number of monomers left of the middle, and it contains a singular element in each slot, namely the direction the next monomer is, relative to where one came from. 0 means straight forward, 1 means to the right and 3 means to the left. Similarly, `afterMiddle` does the same for the right side, just that 2 means straight forward, 1 means to the left and 3 means to the right. Finally, we have also assigned `Middle` with a map, which maps the direction number to cartesian coordinates.

This new implementation speeds up the rotation algorithm, as instead of changing on average $N/2$ datapoints, we only need to rotate one "pointer", as then the relative distance for all points more extreme will also be moved. Furthermore, this also keeps the invaluable property that the middle always is in the center, as all pointers point from the center to the extremes. This implementation however, relies on changing the polymer in-place, which was made possible by implementing an else in the `rotateManyTimes`. If the new polymer wasn't legal, one would just rotate the polymer on the same monomer in the other direction. To do things like illustrate the polymer, we just start from the middle and illustrate each end by keeping track of the position we are at, which way we look and which way the pointer wants us to move. Thus, we can always construct the appropriate x- and y-coordinates in linear time total for all monomers, which is more than fast enough as all algorithms that need the x- and y-coordinates use linear time regardless, and tests tell us that the algorithms that take linear time like for instance to check if a polymer is valid, often operate faster with the new implementation. Without `@jit`, this new implementation halves the computational time of `rotateNTimes`.

To illustrate this, we reimplemented the three necessary functions to `rotateNTimes` with the old implementation without `@jit`. They are identical to the functions above, just without comments, `@jit` and with new names.

[ ]:
```
def validPolymer1(polymer, N):
    if len(polymer) != N:
        return False
    coordinateSet = set()
    coordinateSet.add((polymer[0, 0], polymer[0, 1]))

    for monomerNumber in range(1, N):

        if (polymer[monomerNumber, 0], polymer[monomerNumber, 1]) in␣
↪coordinateSet:
            return False
        else:
            coordinateSet.add((polymer[monomerNumber, 0],␣
↪polymer[monomerNumber, 1]))
        xDiff = np.abs(polymer[monomerNumber, 0] - polymer[monomerNumber - 1,␣
↪0])
        yDiff = np.abs(polymer[monomerNumber, 1] - polymer[monomerNumber - 1,␣
↪1])
```

```
        if xDiff + yDiff != 1:
            return False
    return True

def rotatePolymer1(polymer, monomer, positiveDirection):
    monomer -= 1
    middleMonomer = len(polymer) // 2
    x, y = polymer[monomer]
    newPolymer = np.zeros((len(polymer),2))

    if middleMonomer > monomer:
        newPolymer[monomer:] = polymer[monomer:]
        newPolymer[:monomer, 0] = (2 * positiveDirection - 1) * (polymer[:
 ↪monomer, 1] - y) + x
        newPolymer[:monomer, 1] = (1 - 2 * positiveDirection) * (polymer[:
 ↪monomer, 0] - x) + y
        return newPolymer
    newPolymer[:monomer+1] = polymer[:monomer+1]
    newPolymer[monomer+1:,0] = (1-2*positiveDirection)*(polymer[monomer+1:
 ↪,1]-y)+x
    newPolymer[monomer+1:,1] = (2*positiveDirection-1)*(polymer[monomer+1:
 ↪,0]-x)+y
    return newPolymer

def rotateNTimes1(N, Ns):
    rotationsMade = 0
    polymer = createPolymer(N)
    for i in range(Ns):
        monomer = np.random.randint(2, N)
        positivRetning = np.random.randint(0, 2)

        twistedPolymer = rotatePolymer1(polymer, monomer, positivRetning)
        if validPolymer1(twistedPolymer, N):
            rotationsMade += 1
            polymer = twistedPolymer
    return polymer, rotationsMade
```

And here is the implementation of the functions above with the new polymer representation.

[ ]:
```
class Middle:
    def __init__(self, x, y, N):
        self.position = np.array([x,y]) # Position of the middle
        self.beforeMiddle = np.array([0 for i in range(N//2)]) # Left tail of␣
 ↪the polymer from the middle
        self.afterMiddle = np.array([2 for i in range((N-1)//2)]) # Right tail␣
 ↪of the polymer from the middle
```

```
        self.map = {0:np.array([0,-1]), 1:np.array([1,0]), 2:np.array([0,1]),3:
 ↪np.array([-1,0])} # Map of direction to Cartesian-coordinates change

"""
1 b) This is a new createPolymer designed around the new implementation of a␣
 ↪polymer
"""
def createPolymerVer2(N):
    middle = Middle(N//2, N//2, N)    # Initializes the class
    return middle
"""
1 d) This is a new way to illustrate the polymer using the same principles as␣
 ↪above but with the new implementation in mind
"""
def illustrationPolymerVer2(polymer):
    """
    Illustrating the polymer with the new representation. We could have␣
 ↪implemented the illustration as above, with zoomed in, but as the focus
    of this new implementation is to make the program faster, and as we later␣
 ↪conclude to abandon this version, it was concluded that a proof of
    concept, that one could illustrate the second polymer implementation␣
 ↪relatively efficient, was sufficient.


    Input:
    polymer: The polymer to illustrate

    As the endpoints aren't included in the two arrays in the implementation of␣
 ↪the polymer, but the middle is included twice, the total length
    should be the sum of the lengths of the arrays plus 1.
    """
    N = len(polymer.beforeMiddle) + len(polymer.afterMiddle) + 1
    """
    Initializing the grid and setting the default value to be -N/2 rounded down
    """
    grid = np.zeros((N+1,N+1))
    grid -= N//2
    index = N//2
    position = np.copy(polymer.position)
    direction = 0
    """
    Traversing the left tail of the polymer from the middle, and setting the␣
 ↪value in the grid as the monomer number. The starting direction is
    0 (to the left)
    """
    for firstMonomers in range(index-1,-1,-1):
```

```
        """
        Updating the direction to traverse, realising that as the default is to the right, upwards should decrease one from the direction and
        downwards should add one. Then traversing one that direction
        """
        direction = (direction + polymer.beforeMiddle[firstMonomers])%4
        position += polymer.map[direction]
        grid[position[0],position[1]] = firstMonomers + 1
    direction = 2
    position = np.copy(polymer.position)
    """
    Setting the grid value for the middle monomer
    """
    grid[position[0],position[1]] = index + 1
    """
    Traversing the right tail of the polymer from the middle, and setting the value in the grid as the monomer number. The starting direction is
    2 (to the right)
    """
    for secondMonomers in range(0, len(polymer.afterMiddle)):
        """
        Traversing the right tail of the polymer from the middle, and setting the value in the grid as the monomer number. The starting direction is
        2 (to the right)
        """
        direction = (direction + polymer.afterMiddle[secondMonomers]-2)%4
        position += polymer.map[direction]
        grid[position[0],position[1]] = secondMonomers+ index + 2
    plt.pcolormesh(grid)
    plt.show()

"""
1 e) Checks if the polymer is intact and legal
"""

def validPolymerVer2(polymer, N):
    """
    Checking if the polymer is valid, given the restrictions presented in the problemset
    Input:
    polymer: The polymer to check
    N: The supposed length of the polymer

    Checking if the monomer with monomer number m are closest neighbours with the monomers with monomer numbers m+1 and m-1 is unnecessary,
    since this version of representation is based on telling the monomers which direction they are supposed to go 1 unitlength in.
```

```
    Here we check whether the polymer is of correct length, as the two arrays␣
↪doesnt account for the end points but also including the middle
    twice, the total length of the two arrays should be 1 less than N
    """
    if len(polymer.beforeMiddle) + len(polymer.afterMiddle) + 1 != N:
        return False
    """
    Create a set to store the coordinates of the monomers. The datastructure␣
↪set is chosen because of the O(1) average look up time compared to
    the O(n) of for instance a list. The middle is added to the set
    """
    coordinateSet = set()
    coordinateSet.add((polymer.position[0], polymer.position[1]))
    index = N//2
    position = np.copy(polymer.position)
    direction = 0
    """
    Checking the left tail of the array first. Thus, we start looking to the␣
↪left direction and traversing outwards.
    """
    for firstMonomers in range(index-1,-1,-1):
        """
        Checking if the direction of traversing is valid
        """
        if polymer.beforeMiddle[firstMonomers] not in polymer.map:
            return False
        """
        Updating the direction to look and traversing that direction
        """
        direction = (direction + polymer.beforeMiddle[firstMonomers])%4
        position += polymer.map[direction]
        """
        If the new coordinate is in the set, abort as we have an overlap, else,␣
↪add the new coordinate to the set
        """
        if (position[0],position[1]) in coordinateSet: #Checks if there exists␣
↪an overlap of coordinates
            return False
        else:
            coordinateSet.add((position[0],position[1]))
    direction = 2
    position = np.copy(polymer.position)
    """
    Initialising for the right tail, with starting direction to the right (2)
    """
    for secondMonomers in range(0,len(polymer.afterMiddle)):
```

```
        """
        If the direction is invalid, return false
        """
        if polymer.afterMiddle[secondMonomers] not in polymer.map:
            return False
        """
        Updating the direction to traverse, realising that as the default is to␣
 ↪the right, upwards should decrease one from the direction and
        downwards should add one. Then traversing one that direction
        """
        direction = (direction + polymer.afterMiddle[secondMonomers]-2)%4
        position += polymer.map[direction]
        """
        If the new coordinates is in the set, return false as there is an␣
 ↪overlap, else add the coordinate to the set
        """
        if (position[0],position[1]) in coordinateSet: #Checks if there exists␣
 ↪an overlap of coordinates
            return False
        else:
            coordinateSet.add((position[0],position[1]))
    """
    If all conditions passed, then the polymer must be valid
    """
    return True

"""
1 f) Implementation of rotation, as previously, the middle rounded upwards is␣
 ↪kept at a constant position
"""
def rotationGoBrrrrVer2(polymer, monomer, positivRetning):
    """
    Note that neither the first nor the last monomer in the polymer is␣
 ↪represented in the array, and that also we call the monomers by 1-indexing
    Thus we need to subtract two to both account for the first monomer not␣
 ↪being in the array, and also for that we go from 1-indexing to 0-indexing
    """
    monomer -= 2
    middleMonomer = len(polymer.beforeMiddle)-1 #Finds the middle rounded up
    if middleMonomer > monomer: #Checks if we are before the middle
        polymer.beforeMiddle[monomer] = (polymer.beforeMiddle[monomer] +␣
 ↪2*positivRetning-1) % 4
        """
        Positive direction:
        direction = direction + 1
        Negative Direction:
```

```
        direction = direction - 1

        Then readjust so that for instance right from pointing downwards is␣
 ↪left, not an undefined value

        Follows from that the direction number increases in the negative␣
 ↪direction
        """
        return polymer
    polymer.afterMiddle[monomer-middleMonomer] = (polymer.
 ↪afterMiddle[monomer-middleMonomer] - 2*positivRetning+1) % 4
    """
    Positive direction:
    direction = direction - 1
    Negative Direction:
    direction = direction + 1

    Then readjust so that for instance right from pointing downwards is left,␣
 ↪not an undefined value

    Follows from that the direction number increases in the negative direction
    """
    return polymer

"""
1 g) This is the reimplementation of the similar algorithm above, with the key␣
 ↪difference being what happens when a rotation doesn't work.
Here we undo the rotation by giving the same polymer and monomer, just the␣
 ↪opposite rotation direction. This is necessary as we edit the polymer
in place, and thus can't just discard the changes.
"""
def rotateNTimesVer2(N, Ns):
    """
    Rotate a polymer of length N Ns times
    Input:
    N: The length of the polymer
    Ns: The number of attempted rotations
    """
    rotationsMade = 0
    polymer = createPolymerVer2(N)
    """
    Attempt Ns rotations
    """
    for i in range(Ns):
        """
        Randomly choose a monomer to rotate and a direction to rotate it in
```

```
        """
        monomer = np.random.randint(2, N)
        positivRetning = np.random.randint(0,2)

        polymer = rotationGoBrrrrVer2(polymer, monomer, positivRetning)
        """
        If the new polymer is valid, add another rotation, else, rotate it back
        """
        if validPolymerVer2(polymer, N):
            rotationsMade += 1
        else:
            polymer = rotationGoBrrrrVer2(polymer,monomer,(positivRetning+1)%2)

    return polymer, rotationsMade

"""
Here is a comparrison of the old rotateNTimes with jit, rotateNTimesVer2 and␣
 ↪the old rotateNTimes without jit.
"""
print("rotateNTimes with jit")
%timeit rotateNTimes(15, 1000)
print("New implementation")
%timeit rotateNTimesVer2(15,1000)
print("rotateNTimes without jit")
%timeit rotateNTimes1(15,1000)
```

```
rotateNTimes with jit
4.17 ms ± 33.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
New implementation
12.2 ms ± 112 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
rotateNTimes without jit
24.5 ms ± 931 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

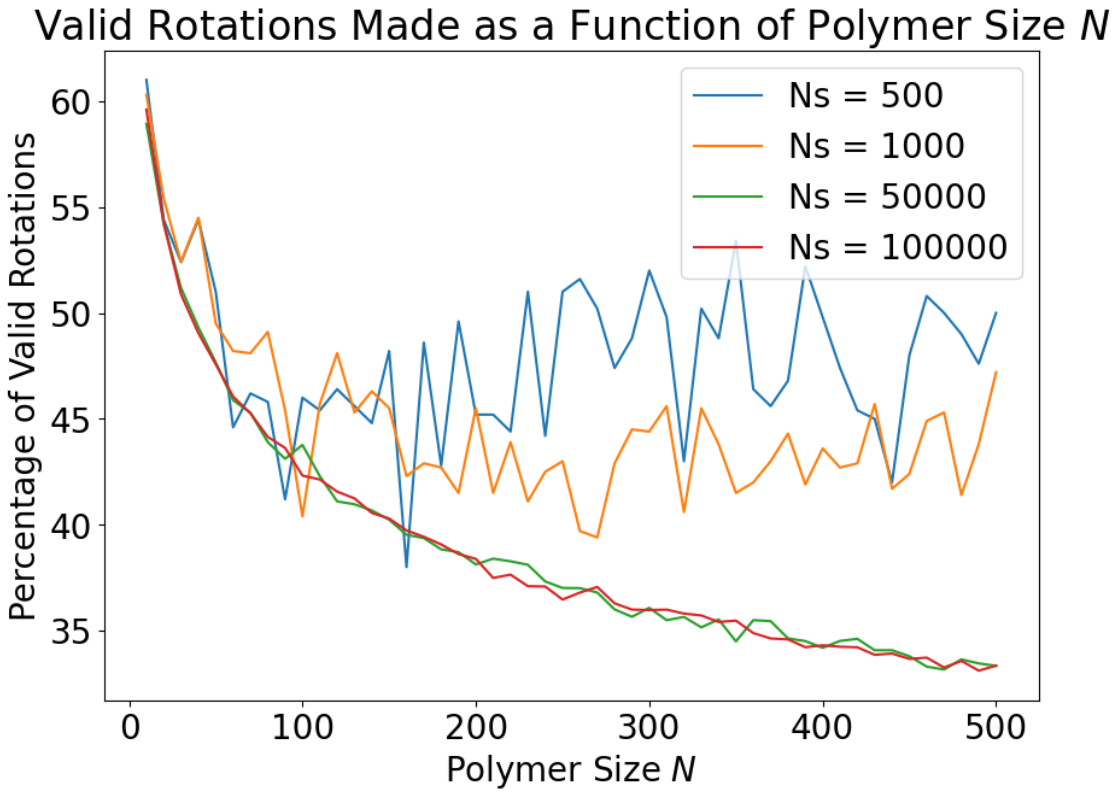
A quick analysis of `rotateNTimes` shows that its time complexity is $O(N \cdot N_s)$, as both `validPolymer` and `rotationGoBrrrrVer2` run with timecomplexity $O(n)$, and they are run $\theta(N_s)$ times each. Thus, `rotateNTimes` should scale linearly with both increase in polymer size and increase in number of rotations.

Furthermore, as discussed above, we can see that the new implementation is faster, when both algorithms are run without `@jit`, which is mostly due to the fact that rotation of the polymer is much faster. However, as we have been able to make `@jit` work with the old implementation and not the new implementation, and that the old implementation is much more clear and sufficiently quick, we have chosen to abandon the second implementation, and continue focusing on the old implementation.

The function `rotateNTimes` receives the argument `Ns`, which is the number of rotations we wish to complete. However, since we encounter several non-valid polymers while rotating many times, we want to see how many of those `Ns` rotations actually resulted in valid polymers. `rotateNTimes` returns `rotationsMade`, which tells us how many rotations resulted in a valid polymer. The

following function, as described in **1i)**, plots the percentage of valid rotations as a function of the size N of the polymer. `plotValidPercentage` receives `minSize` and `maxSize`, which are the minimum and maximum sizes of the polymers we are to rotate. `minSize` and `maxSize` are by default set to 10 and 500, respectively. `plotValidPercentage` also has the input `NsArray`, which is an array of the different `Ns` we want to test. `NsArray` is default set to 1000 rotations.

[ ]:
```
def plotValidPercentage(minSize = 10, maxSize = 500, NsArray = np.
↪array([1000])):
    """
    Plotting the valid percentages as a function of polymer lengths for␣
↪different amounts of attempted rotations
    Input:
    minSize: The shortest polymer to simulate for a range with interval-sizes␣
↪of 10
    maxSize: The longest polymer to simulate for the range
    NsArray: The different number of rotations attempted, each of these will be␣
↪used to plot percentage of successfull rotations as a function of polymer␣
↪size
    """
    sizes = np.arange(minSize, maxSize + 1, 10)
    intSizes = sizes.astype(int)
    """
    Initialising the plot
    """
    plt.figure(figsize = (10, 7))
    """
    Simulating the rotations and plotting the percentage of valid rotations
    """
    for Ns in NsArray:
        validRotations = np.array([rotateNTimes(N, Ns)[1] for N in intSizes])
        plt.plot(intSizes, (validRotations / Ns) * 100, label = f'Ns = {Ns}')
    """
    Customizing the plot
    """
    plt.xlabel(r'Polymer Size $N$')
    plt.ylabel('Percentage of Valid Rotations')
    plt.title('Valid Rotations Made as a Function of Polymer Size $N$')
    plt.legend()
    plt.show()

plotValidPercentage(10, 500, np.array([500, 1000, 50000, 100000]))
```



The first thing that we observe in the plot is that larger N_s makes the curve smoother and with less variation. Secondly, the plot seems to indicate that all N_s seems to have a saturation N, from which the percentage of valid rotations seems to stabilize with variation around a given percentage for all polymers length greater than N. This saturation percentage seems to be lower for greater N_s and the saturation length seems to be greater for greater N_s. For both `Ns=500` and `Ns=1000`, the saturation seems to be around `N=50` and with percentages approxiemtly 47.5% and approximetly 45% respectively, while for `Ns=50000` and `Ns=100000`, `N=500` doesnt seem sufficient to find this saturation point.

As described in task **1j)**, we now implement energy calculations. The following function `calculateEnergy` receives a polymer and a matrix V. The matrix V is, as described in the project description, a symmetric matrix that contains the strengths of the interactions between monomers. The strength of the interaction is 0 for the interaction between a monomer m and the monomers $m-1$, m, $m+1$, and -1 elsewhere in this example.

To calculate the energy of each polymer, there are three major ideas behind our design `calculateEnergy`. Firstly, we have realised that if monomers x and y are neighbours, with $y > x$ and $x, y \in \mathbb{Z}^+$, we only need to catch that y is a neighbour to x. This is because the potential is symmetrical, and thus instead of adding half of the potential for each of them, we just add the entire potential when we discover that y is a neighbour to x. Secondly, we discovered that it doesn't matter that we discover that m is a neighbour with $m-1$, as adding the potential between them will just be adding 0. Thirdly, we discovered that it doesn't matter if we store all potential neighbours to a coordinate as a list and loop over them, as a point can maximum have 4 neighbours,

and thus one only needs to loop over 4 values regardless, which is constant itme. Thus, our design method of the algorithm is that we start at the beginning of the polymer, and for each monomer, we log which coordinates the monomer is neighbour to, and save these data in a dictionary where the key is a coordinate, and the values are the monomers discovered so far that are neighbours to this coordinate. As dictionaries are hash-tables in python, each coordinate has finite neighbours and each monomer has finite coordinate-neighbours, each iteration over a monomer should take constant time, and calculateEnergy should average $\theta(n)$, which seems reasonable to be as efficient in asymptotic notation as this algorithm can be.

[ ]:
```
def calculateEnergy(polymer, V):
    """
    Calculate the energy of the polymers
    Input:
    polymer: The polymer to calculate the energy for
    V: The potential used to calculate the energy
    """
    energy = 0
    neighbourDict = {}
    direction = [[0, 1], [0, -1], [1, 0], [-1, 0]]
    """
    Loop over the monomers in the polymers, and checking if they are neighbours
    """
    for monomerNumber, coordinates in enumerate(polymer):
        cordTuple = (coordinates[0], coordinates[1])
        """
        Checking if the coordinates of the monomer makes it neighbour with any␣
↪previously checked monomers.
        """
        if cordTuple in neighbourDict:                              #
            for n in neighbourDict[cordTuple]:
                energy += V[monomerNumber,n]
        """
        Adding the neighbouring spaces of the monomer to the␣
↪neighbourdictionary, in case one of the later monomers are on one of these␣
↪spots
        """
        for vec in direction:
            temp = (coordinates[0] + vec[0], coordinates[1] + vec[1])

            if temp in neighbourDict:
                neighbourDict[temp].append(monomerNumber)
            else:
                neighbourDict[temp] = [monomerNumber]
    """
    Returning the energy
    """
    return energy
```

We want to use this function to calculate the energy of the two polymers from task **1g)**. As task **1j)** asks, we use a V-matrix where all non-zero elements are -1. First, we define the function `makePotential`, to easily make the matrix V.

[ ]:
```
def makePotential(N, backgroundValue):
    """
    Make a potential matrix:
    Input:
    N: The size of the polymer, and thus the length of each side of the␣
 ↪potential matrix
    backgroundValue: The value of all cells not supposed to be 0 (the cells␣
 ↪representing the interaction of monomer m with monomers m-1,m,m+1)
    """
    V = np.zeros((N, N)) + backgroundValue
    """
    Setting the correct cells to 0
    """
    for m in range(N):
        V[m, m] = 0
        if m > 0:
            V[m, m - 1] = 0
            V[m - 1, m] = 0

    return V
```

[ ]:
```
"""
From Task 1g)
"""

print("From task 1g)\n")

VN4 = makePotential(len(polymerN4), backgroundValue=-1)

print(f'The energy of the polymer with 4 attempts of rotation is␣
 ↪{calculateEnergy(polymerN4, VN4)}.')

VN1000 = makePotential(len(polymerN1000), backgroundValue=-1)

print(f'The energy of the polymer with 1000 attempts of rotation is␣
 ↪{calculateEnergy(polymerN1000, VN1000)}.')
```

```
From task 1g)

The energy of the polymer with 4 attempts of rotation is 0.0.
The energy of the polymer with 1000 attempts of rotation is -2.0.
```

The polymer with 4 tries of rotation often has a smaller absolute energy than the polymer with 1000 tries of rotation. This does intuitively make sense, since 4 rotations often isn't enough to

curl the polymer into something ball-like, that has a lot of neighbours. 1000 is plenty though, and even though it wouldn't randomly make a ball, it does seem intuitive that segments of the polymer would seem cramped up, giving lots of neighbours, for example if there are 2 consecutive positive rotations next to each. We would however expect to see quite a lot of cases where the opposite is true, since the 1000 rotations often can counteract each other.

1.0.2 Monte Carlo Simulations

That concludes the first part of the project, i.e. implementing polymer functionality. Moving on to part 2, this functionality will be used to create realistic rotations and investigate energy at different temperatures. Now that we have a function that can calculate the energy of a given polymer, we can start to simulate a polymer through a Monte Carlo simulation. The Monte Carlo simulation, as described in algorithm 2, takes the fact that a polymer want to achieve lower energy into account, as well as considering energy rises due to thermal fluctuations.

Before implementing algorithm 2 in the function `metropolisAlgorithm`, we start by implementing a function that calculates a polymers diameter. This is done considering a later task, **2g)**, where we are to add diameter calculations to our implementation of algorithm 2. The function `calculateDiameter` finds the longest cartesian distance between two monomers of the polymer. Thus, for instance, a polymer that's of dimensions $1 \times N$ will get a diameter of $N-1$ as the longest distance would be between the midpoints of the edge monomers. `calculateDiameter` has only one argument, which is a polymer. Our algorithm checks the distance between all pair of monomers and returns the largest distance. This is not a time efficient algorithm, we therefore use `@jit` again to reduce the runtime.

[ ]:
```
@jit
def calculateDiameter(polymer):
    """
    Calculate the diameter of the polymer:
    Input:
    polymer: The polymer to calculate the diameter of
    """
    maxDist = 0

    """
    Find the distance between each pair and return the maximum one.
    As the square root function is bijective, finding the maximum without it is␣
 ↪the same as finding the maximum with applying it, so to save
    computation time, we only take the square root of the answer
    """
    for i in range(len(polymer)):
        for j in range(i + 1, len(polymer)):
            distance = np.sum((polymer[i] - polymer[j]) ** 2)

            if distance > maxDist:
                maxDist = distance

    return np.sqrt(maxDist)
```

As instructed in task **2a)**, we now implement algorithm 2. `metropolisAlgorithm` takes in a `polymer`, the potential matrix `V`, the number of rotations `Ns` and the temperature `T`. To answer task **2g)** we also added the boolean argument `includeDiameter`, which is default set to `False`, so we don't have to calculate the diameter each time we use the `metropolisAlgorithm`.

[ ]:
```
k_b = 1.38 * 10 ** (-23)            # Boltzmanns constant

def metropolisAlgorithm(polymer, V, Ns, T, includeDiameter = False):
    """
    Implementation of algorithm 2 with diameter calculations:
    Input:
    polymer: The polymer to simulate
    V: The potential matrix to calculate the energy of the polymer
    Ns: The number of successfull rotations needed
    T: The temperature at which the simulation takes place
    includeDiameter: Whether the metropolisAlgorithm should calculate the␣
↪diameters or not
    """
    E_array = np.zeros(Ns)
    E = calculateEnergy(polymer, V) # Calculate the initial energy of the␣
↪polymer

    if includeDiameter:
        d_array = np.zeros(Ns)
        d = calculateDiameter(polymer) # If relevant, calculate the intital␣
↪diameter of the polymer
        d_array[0] = d

    """
    Setting constants to help further down in the process
    """
    rotationsMade = 0
    N = len(polymer)
    beta = 1 / (k_b * T)
    E_array[0] = E

    """
    The main body of the algorithm, continue until Ns sucessfull rotations␣
↪achieved
    """
    while rotationsMade < Ns - 1:
        newpolymer = rotatePolymer(polymer, np.random.randint(2, N), np.random.
↪randint(0,2)) #Simulate a rotation

        if validPolymer(newpolymer,N):
            """
            Valid rotation achieved, check if the polymer should change
```

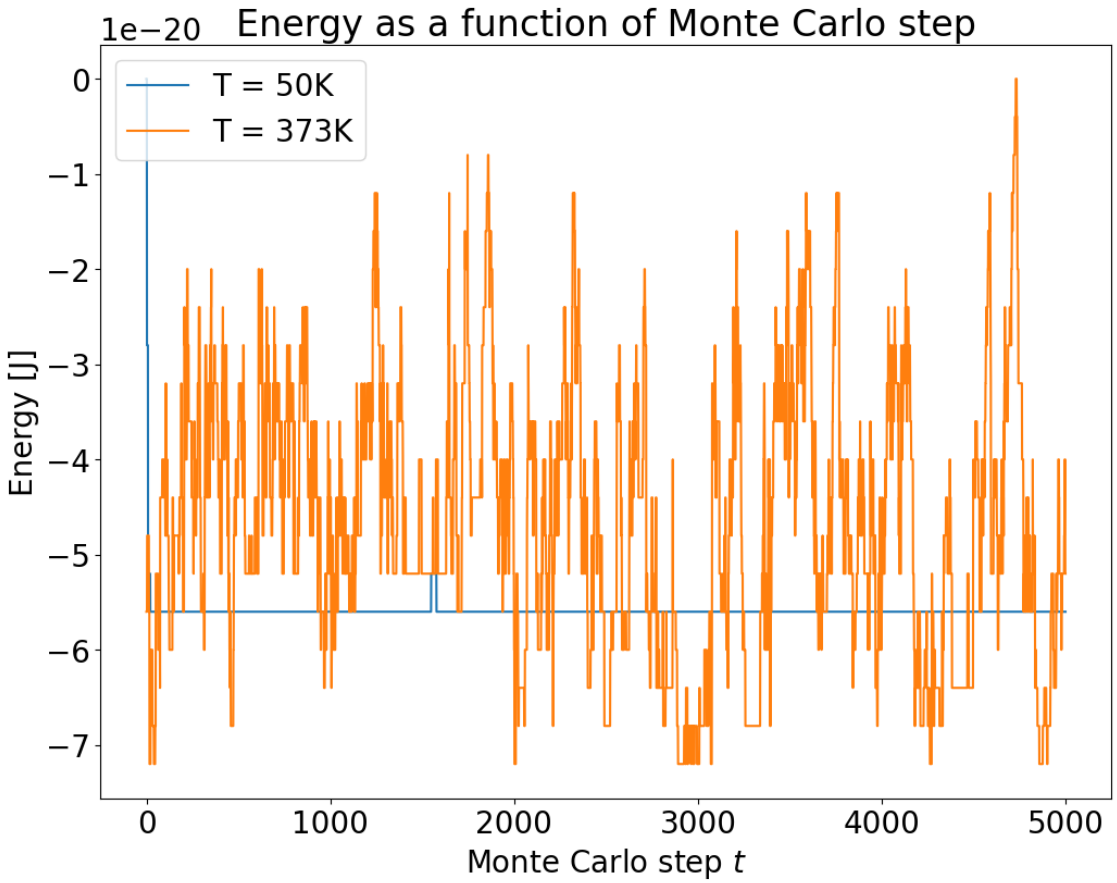
```
            """
            rotationsMade += 1
            E_new = calculateEnergy(newpolymer, V)

            if includeDiameter:
                d_new = calculateDiameter(newpolymer)

            if E_new < E:
                """
                Update the polymer if the new polymer has less energy
                """
                polymer = newpolymer
                E = E_new

                if includeDiameter:
                    d = d_new

            elif np.random.uniform() < np.exp(-beta * (E_new - E)):
                """
                Update the polymer if the new polymer has more energy, but␣
↪probability dictates that we should update regardless
                """
                polymer = newpolymer
                E = E_new

                if includeDiameter:
                    d = d_new
            """
            Insert the newest value of the energy (and possible the diameter)␣
↪into the respective arrays
            """
            E_array[rotationsMade] = E
            if includeDiameter:
                d_array[rotationsMade] = d

    """
    Return the data
    """
    if includeDiameter:
        return polymer, E_array, d_array

    return polymer, E_array
```

As mentioned, the Monte Carlo simulation makes polymers search for lower energy, as well as considering that the energy may increase due to thermal fluctuations. At higher temperatures, there are stronger thermal fluctuations, so the energy is more likely to increase than at lower temperatures. To see how these thermal fluctuations affect the energy, we will compare a polymer

at a low temperature $T = 50K$ to a polymer at a high temperature $T = 373K$. We will compare this by plotting the energy of the two polymers as a function of the number of Monte Carlo steps, i.e. the number of rotations that resulted in a valid polymer.

We want to see how the polymers energy varies as a function of the number of Monte Carlo steps. The following function plots said energy for a low temperature $T = 50K$ and a higher temperature $T = 373K$. To work with these polymers later in the code, `plotEnergy` returns the polymers simulated.

[ ]:
```
def plotEnergy(polymer, V, Ns, T_array):
    """
    Plot the energy of the polymers simulated with the metropolisAlgorithm for␣
 ↪different temperatures
    Input:
    polymer: The polymer to simulate the metropolisAlgorithm on with various␣
 ↪temperatures
    V: The potential matrix for the metropolisAlgorithm
    NS: The number of successfull rotations for the metropolisAlgorithm
    T_array: The temperatures to simulate the metropolisAlgorithm on
    """
    polymers = []
    plt.figure(figsize = (12, 9))
    """
    Plotting the energy-arrays for each temperature
    """
    for T in T_array:
        polymer, E_array = metropolisAlgorithm(polymer, V, Ns, T)
        polymers.append(polymer)
        plt.plot(E_array, label = f'T = {T}K')
    """
    Customizing the plot
    """
    plt.xlabel(r'Monte Carlo step $t$')
    plt.ylabel(r'Energy [J]')
    plt.title('Energy as a function of Monte Carlo step')
    plt.legend()
    plt.show()
    """
    Returnin the achieved polymers for a later problem
    """
    return polymers

N = 30
V = makePotential(30, -4 * 10 ** (-21))
polymerLowTemp, polymerHighTemp = plotEnergy(createPolymer(N), V, 5000, [50,␣
 ↪373])
```

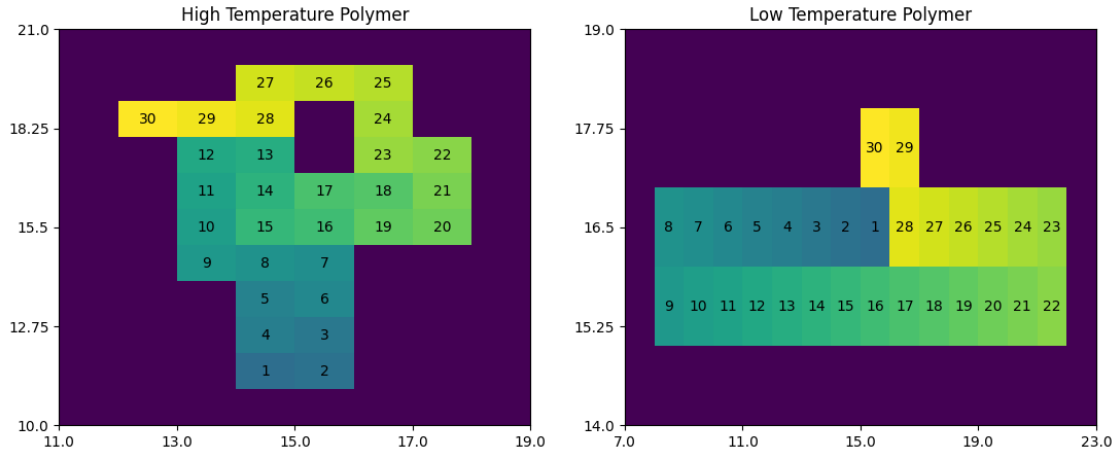


The plot shows that the energy for the polymer at a lower temperature $T = 50K$ often varies less than the energy of the polymer at the higher temperature $T = 373K$. The polymer at $T = 50K$ has a quick drop in the energy after a couple Monte Carlo steps, before it stabilizes at a local energy minimum. In some simulations, the polymers manages to break out of the local energy minimum and the energy drops further to another local energy minimum.

The polymer at $T = 373K$, starts oscillating. Even though the polymer at a higher temperature reaches lower energies throughout the simulation, the polymer at lower temperatures often has a lower energy average, as it stabilizes at a low energy instead of oscillating between lower and higher energies. This makes sense, since temperature is just a measurement of movement, and higher temperatures often indicate more possible states.

The following code block illustrates how these two polymers look after the simulation, as mentioned in task **2c)**. Note that the illustration code for this task is implemented earlier, as discussed previously.

[ ]:
```
plt.rcParams.update({'font.size': 10}) # change font-size of the plot
multiplePlotsPolymers(polymerHighTemp, polymerLowTemp, "High Temperature␣
↪Polymer", "Low Temperature Polymer")
```



In the plot of the low energy polymer, which is the case most of the time one simulates a low temperature polymer, one can observe that everything is neetly packed together and that most if not all legal rotations will significantly increase the energy of the polymer. This makes sense, as the temperature is low, and thus the probability of the polymer doing an energetically unfavoured rotation is slim. Thus, it will most likely just slowly transition as far down into the local energy minima and stay there, similar to how the graph looks with it being stable at a low energy. The high temperature polymer on the other hand, is usually way less compact, as can be observed in the above graph. This also makes sense, as which a high temperature, the probability of doing an energetical unfavourable twist is more significant. Thus, the high temperature polymer will to greater degree not seek a local minima compact representation, which match with the two illustrations above.

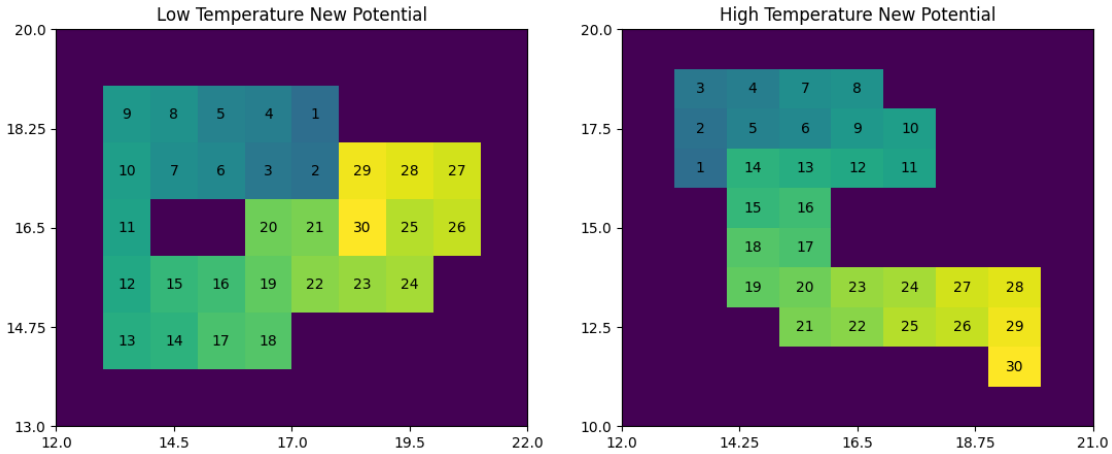
Until now, we've looked at potential matrices V where all non-zero values are the same. As instructed in task **2d)**, the following function `makeSymmetricPotential` creates a symmetrical potential matrix, which follows that the potential between all monomers m and monomers $m-1,\ m,\ m+1$ are 0. Thus, it creates a $N \times N$ array and assigns the general background value. Then, it loops through a list of pairs, of which are supposed to have a potential equal to the general value, multiplied by a scaling factor, thus creating a non-uniform potential. Finally, it puts the potential between a monomer m and the monomers $m-1,\ m,\ m+1$ to 0. The final step is done last, to ensure that in case for instance the tuple $(4, 3)$ is included, that pair still has 0 potential between them.

To test how such a potential matrix will change the outcome, we have chosen to scale all pairs of the form $(i, i+3)$ by a factor of 100. At low temperature, it is hypothesised that little will change, as the polymers struggle to do energetically un-favoured transitions, and this will just amplify the differences between different states and thus restrict the polymers ability to do such transitions. However, it probably will make it slightly more zig-zag, which is how the pairs $(i, i+3)$ become neighbours, as two states that used to be energetically identical before heavily favours the one with $(i, i+3)$ as neighbours. For large temperatures, on the other hand, it is hypothesised that the difference will be huge. Ths is due to the fact that normally, it is quite easy for polymers at these temperatures to do transitions that increase a polymers energy. However, mathematically, one can see that a polymer breaking a $(i, i+3)$ bond at 300K is as likely as a polymer at 3K breaking a

bond, which is very unlikely. Thus, the the polymer at 300K should fold itself in a zig-zag pattern, where many pairs of $(i, i+3)$ become neighbours.

[ ]:
```
def makeSymmetricPotential(N, generalValue, scaling, tuplesToScale):
    """
    Input:
    N: The length of the polymer
    generalValue: the value of the cells which are neither 0 nor the interactions which are to be scaled
    scaling: the factor which the generalValue is multiplied with for the specified pairs
    tuplesToScale: the pairs which are to have a difference potential between them than the general value
    """

    potential = np.zeros((N,N)) + generalValue # Set the backround value
    """
    Scaling the tuples to scale, and ensuring that the matrix stays symmetric
    """
    for tup in tuplesToScale:
        potential[tup[0],tup[1]] = generalValue*scaling
        potential[tup[1],tup[0]] = generalValue*scaling
    """
    Ensure that the the potential between consecutive monomers and the monomers with themselves is 0
    """
    for i in range(N):
        potential[i,i] = 0
        if i > 0:
            potential[i-1,i] = 0
            potential[i,i-1] = 0
    return potential


N = 30
"""
Create a numpy array which groups all pairs of the form (i,i+3) for a polymer of length N
"""
tuples = np.zeros((N-3,2))
for i in range(N-3):
    tuples[i] = np.array([i,i+3])
"""
Make the potential matrix and simulate one instance for a high temperature and one for a low, and plot the results side by side.
"""
V = makeSymmetricPotential(N,-4*10**(-21),100,tuples.astype(int))
```

```
polHighTemp, array = metropolisAlgorithm(createPolymer(N),V,1000, 300)
polLowTemp, array1 = metropolisAlgorithm(createPolymer(N),V,1000, 30)
multiplePlotsPolymers(polLowTemp,polHighTemp, "Low Temperature New Potential",␣
  ↪"High Temperature New Potential")
plt.rcParams.update({'font.size': 20}) # change font-size of the plot
```



After having a couple of simulations for high and low temperatures with and without the new potential matrix, it seems like the hypothesis was correct. Like suspected, the low temperature polymer has a couple of $(i, i+3)$ neighbours, but it doesnt seem apparent from the graph alone that these pairs have lower potential energy. However, from the high-temperature, it seems quite apparent, which coincides with our hypothesis.

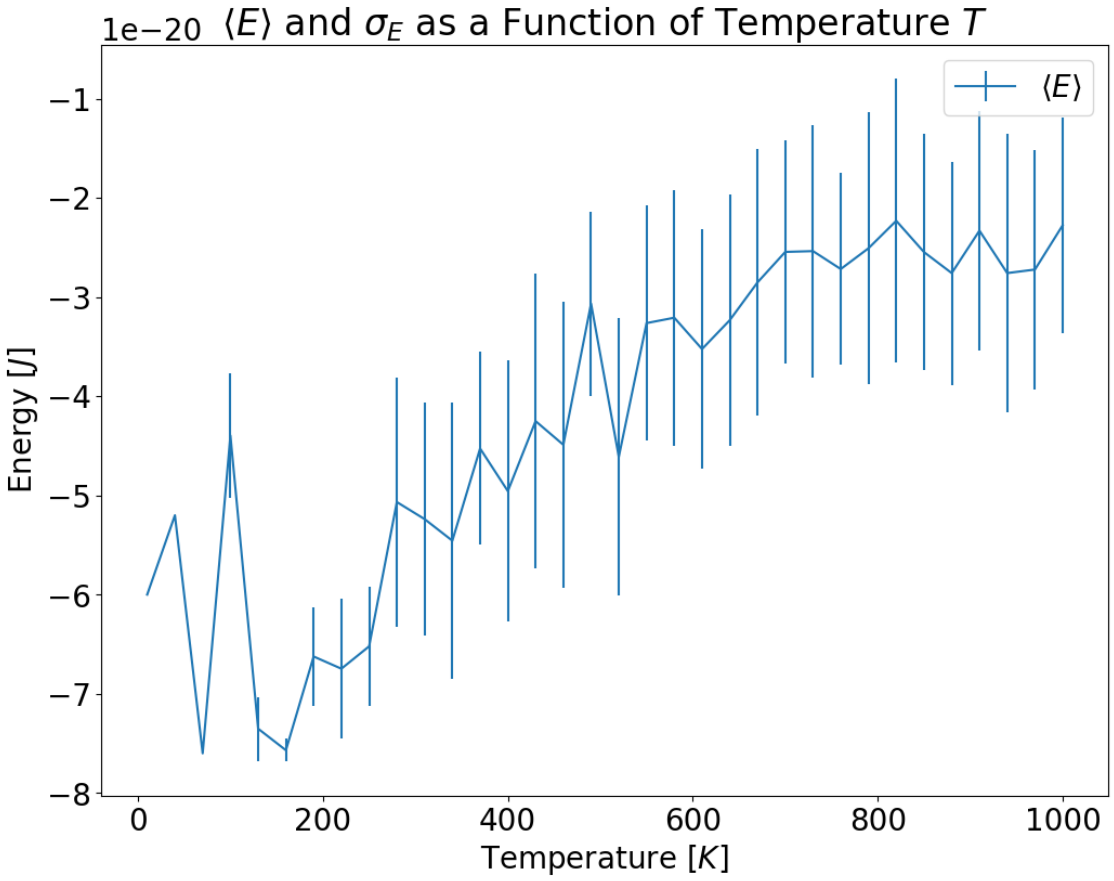
We've seen that temperature is a crucial factor for the energy of a polymer. In the following code block, the expectation value $\langle E \rangle$ and standard deviation σ_E of the energy is shown as a function of the temperature T for polymers of size $N = 30$. For each temperature, a horizontal polymer is created and N_s rotations are completed. The first 1000 steps from each simulation are discarded before calculating $\langle E \rangle$ and σ_E, as specified in task **2e)**.

To accomplish, we made two functions `plotEnergyStats` and `computeEnergyStats`. `computeEnergyStats` takes in a potential matrix, a temperature, number of rotations to accomplish in the `metropolisAlgorithm` and the length of the polymer. It then simulates the `metropolisAlgorithm`, and looks at the energy array that is generated. It discards the first 1000 data-points and takes the average and experimental standard deviation of the remaining energy data, to get a measure of $\langle E \rangle$ and the standard deviation of the measurements for a given temperature.

`plotEnergyStats` takes in a potential matrix, a minimum temperature, a maximum temperature, a temperature step, number of accomplished rotations and the length of a polymer. Then, it creates a temperature range with the minimum and maximum temperatures and the difference between consecutive temperatures as the temperature step. For each of these temperatures, it runs `plotEnergyStats` and stores the expected value and standard deviation. Finally it plots the expected values as a function of the temperatures inputed, before adding the standard deviation as errorbars.

```
[ ]: def computeEnergyStats(V, T, Ns = 1500, N = 30):
        """
        Initialize and simulate the metropolisAlgorithm for a polymer of length N
        Input:
        V: The potential matrix of which to simulate the metropolisAlgorithm on
        T: The temperature at which the metropolisAlgorithm is to take place
        Ns: The number of rotatations to make in the metropolisAlgorithm, must be␣
     ↪over 1000
        N: The length of the polymer
        """
        polymer = createPolymer(N)
        _, energy = metropolisAlgorithm(polymer, V, Ns, T)
        """
        Restricting the energy-data contributing to the expected value and standard␣
     ↪deviation to the values after a 1000, to give the polymer time
        to stabilize
        """
        importantEnergy = energy[1000:]
        return np.average(importantEnergy), np.std(importantEnergy, ddof=1)

    def plotEnergyStats(V, lowTemp, highTemp, tempStep, Ns = 1500, N = 30):
        """
        Plot the expected value and standard deviation of the energy a polymer in a␣
     ↪given range of temperatures

        Input:
        V: The potential matrix to simulate the metropolisAlgorithm on.
        lowTemp: The temperature at which to start the temperature range of which␣
     ↪the metropolisAlgorithm is to be simulated with.
        highTemp: The temperature at which to end the temperature range.
        tempStep: The difference between two consecutive values in the temperature␣
     ↪range
        Ns: The number of succesfull rotations to be done in the␣
     ↪metropolisAlgorithm, must be greater than 1000
        N: The length of the polymer to be simulated
        """

        tempArray = np.arange(lowTemp, highTemp + 1, tempStep)

        """
        Get the expectedvalue and experimental standard deviation for energy as a␣
     ↪function of temperature
        """
        expectedValue, standardDeviation = np.zeros(len(tempArray)), np.
     ↪zeros(len(tempArray))
        for temp_index in range(len(tempArray)):
```

```
        expectedValue[temp_index], standardDeviation[temp_index] =␣
 ↪computeEnergyStats(V, tempArray[temp_index], Ns, N)
    """
    Plot the information
    """
    plt.figure(figsize = (12, 9))
    plt.errorbar(tempArray, expectedValue, yerr = standardDeviation, label =␣
 ↪r"$\langle E\rangle$")
    plt.title(r'$\langle E\rangle$ and $\sigma_E$ as a Function of Temperature␣
 ↪$T$')
    plt.xlabel(r'Temperature [$K$]')
    plt.ylabel(r'Energy [$J$]')
    plt.legend()
    plt.show()

V = makePotential(30,-4 * 10 ** (-21))
plotEnergyStats(V, lowTemp = 10, highTemp = 1000, tempStep = 30, Ns = 1500, N =␣
 ↪30)
```



The above plot shows that the expectation value of the energy $\langle E \rangle$ increases with the temperature T, as guessed. The standard deviation σ_E is much smaller for lower temperatures. This was visualized in the plot where the energies of a low temperature and high temperature polymer were compared. It was clear that for a low temperature, the energy was stabilized, whereas for a higher temperature the energy oscillated. We continue to see this stabilization for low temperature polymers in the next task where we simulate several polymers at lower temperature. The size of σ_E remains seemingly unchanged after temperatures around $300K$ to $400K$.

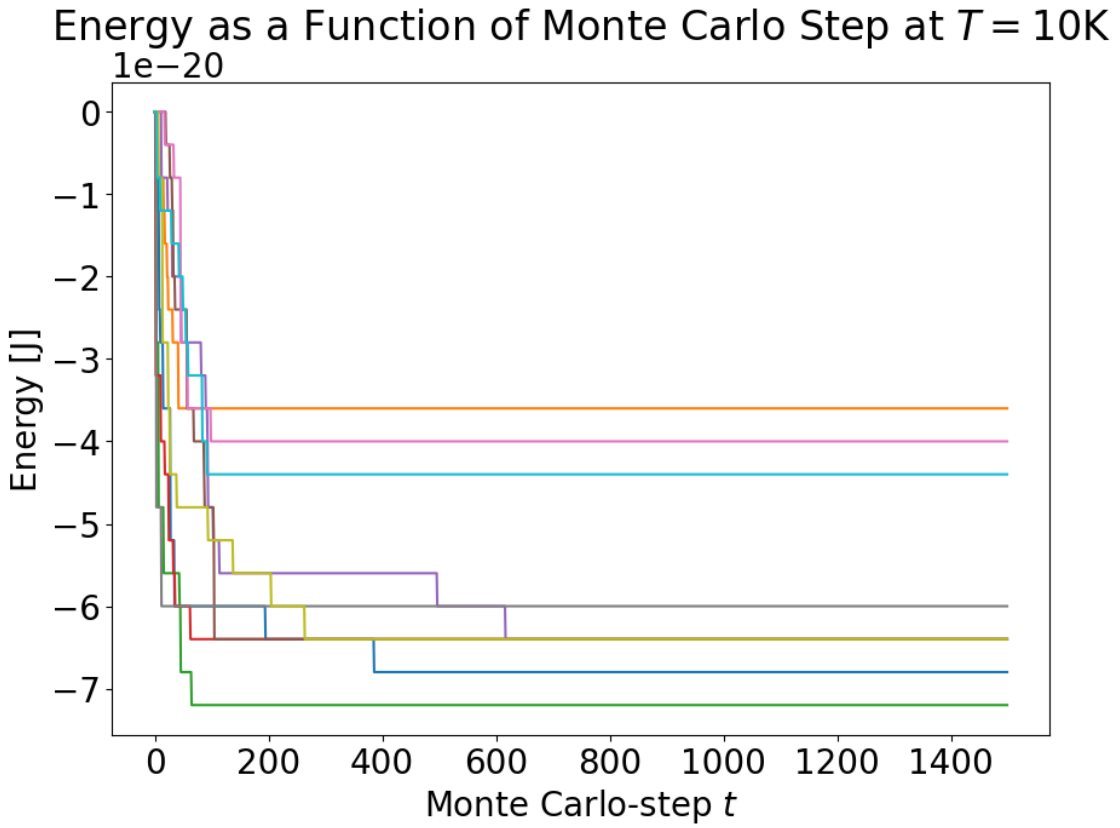
The biggest difference when increasing N_s seems to be that $\langle E \rangle (T)$ seems to be a smoother curve, which ties back to the discussion in **1i)** where larger N_s makes the curve smoother. The minimum, maximum, standard deviation and the general curve of the graph seems to be the same for different N_s.

Another important point we find relevant to this graph is that the standard deviation is for the same polymer, and thus a low energy polymer that has reached an energy minima will have a low standard deviation, while the higher temperature polymers oscillate to a greater degree. However, after having done this simuation quite some time, it seems that the expected energy of the higher temperature polymers stays relatively constant while the low temperature polymers to a greater degree reaches different energies. However, this makes sense, as higher temperature polymers will oscillate between the same values, while low temperature polymers are to a large degree susceptible to which energy minima they reach.

We continue on with the a lower temperature from the preceding plot for problem **2f)**. The following code block simulates $N_s = 1500$ rotations for 10 polymers of length $N = 30$ at a temperature $T = 20K$.

[ ]:
```
def plotEnergyLowTemp(V, T, Ns = 1500, N = 30):
    """
    Comparrison of the energy of different simulation of polymers at the same␣
 ↪length and temperature
    Input:
    V: The potential matrix to simulate with
    T: The temperature to simulate at
    Ns: The number of succesful to complete
    N: The length of the polymers
    """
    plt.figure(figsize = (10, 7))

    """
    Simulate the metropolisAlgorithm 10 times and plotting the energy for each
    """
    for sim in range(10):
        polymer = createPolymer(N)
        _, energy = metropolisAlgorithm(polymer, V, Ns, T)
        plt.plot(energy)

    """
    Fixing the layout of the figure:
    """
```

```
    plt.xlabel(r'Monte Carlo-step $t$')
    plt.ylabel('Energy [J]')
    plt.title('Energy as a Function of Monte Carlo Step at ' + r'$T=10$K')
    plt.show()

V = makePotential(30,-4 * 10 ** (-21))
plotEnergyLowTemp(V, 10, Ns = 1500, N = 30)
```



From the graph, we can see that all 10 simulations behaved relatively similar. One can observe that there are few places where a polymer goes up in energy, if any places at all. Furthermore, one can see that all of them stabilizes relatively uniformly distributed between $-3 \cdot 10^{-20}$J and $-8 \cdot 10^{-20}$J. As all of the polymers behaved so similarly, and all of them ended up in this range, it would be logical that a next simulation of an identical polymer also will have these characteristics. This follows from that if there exists a probability that would push the future polymer in a different direction with traits that oppose the traits shaed by these polymers, it would most likely have appeared in at least one of the 10 simulated polymers. A thing that could potentially occur is that the energy is less than $-8 \cdot 10^{-20}$J or more than $-3 \cdot 10^{-20}$J, however, the more simulations we do that all coincide with the traits discussed here, the more unlikely it is for a future polymer to have traits that contradicts these general traits.

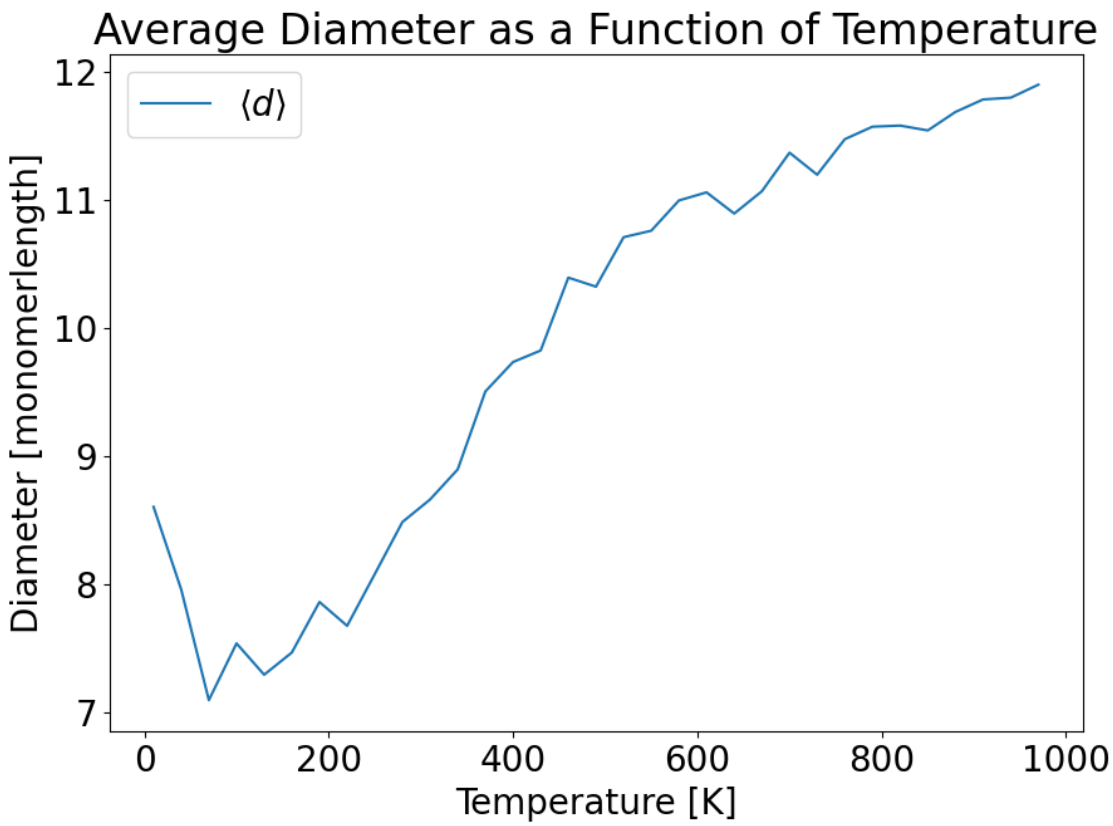
Now, for **2g)** we are going to do something similar to problem **2e)**, just where we consider the

diamater of the polymer, rather than the energy. To accomplish this, we have already implemented `calculateDiameter` along with implementing the `metropolisAlgorithm` in **2a)**, to save us from having to implement `metropolisAlgorithm` twice. Thus this function is explained there, but in general, it just finds the longest cartesian distance from the center of a monomer to the center of another monomer for all pairs of monomers in a polymer. Then, we do the same as for energy in **2e)**, where we create a temperature array, and for each tempearture simulate the `metropolisAlgorithm`, where we for each iteration save the average diameter for the diameter data-points after the first 1000 legal rotaitons. The two major differences between this task and **2e)**, in addition to calculating the average diameter rather than energy, is that we don't calculate the standard deviation, and that the potential matrix contains random values between $-2 \cdot 10^{-21}$J and $-6 \cdot 10^{-20}$J for all non-zero cells. There is still a restriction that the matrix must be symmetrical.

[ ]:
```
def computeDiameterStats(V, T, Ns=1500, N=30):
    """
    Compute the average diameter of the polymer after it has stabilised
    Input:
    V: The potential to simulate on
    T: The temperature at which to simulate
    Ns: The number of successfull rotations to achieve, must be greater than␣
 ↪1000
    N: The length of the polymer
    """
    polymer = createPolymer(N)
    """
    Get the diameter data from the simulation and discard the values before␣
 ↪1000 rotations so the polymer has stabilised a bit
    """
    _,_,diameter = metropolisAlgorithm(polymer, V, Ns, T, includeDiameter=True)
    importantDiameter = diameter[1000:]
    return np.average(importantDiameter)
def getDiameterFromTemp(lowTemp, highTemp, tempStep, Ns=1500, N=30):
    """
    Find expected diameter for a range of temperatures between lowTemp and␣
 ↪highTemp:
    Input:
    lowTemp: The temperature at which the range of temperature to simulate the␣
 ↪metropolisAlgorithm starts
    highTemp: The temperature at which the range ends
    tempStep: The difference between consecutive temperatures in the range
    Ns: The number of successfull rotations for each simulation, must be␣
 ↪greater than 1000
    N: The length of the polymer to simulate

    Create a random symmetric matrix with values between -6*10**(-21) and␣
 ↪-2*10**(-21)
    """
    V = np.zeros((N,N))
```

```
    for i in range(N):
        for j in range(i-1):
    # To ensure that the diagonal and the lines directly above and below are 0
            V[i,j]=(np.random.uniform(-6,-2))*10**(-21)

    V=V+V.transpose()
    # Make it symmetric

    """
    Calculate the exepcted value and standard deviation a range of tempeartures
    """
    tempArray = np.arange(lowTemp,highTemp,tempStep)
    # Create a temperature array which contains the temperatures we are gonna measure
    expectedValue = np.zeros(len(tempArray))
    for index in range(len(tempArray)):
        expectedValue[index] = computeDiameterStats(V, tempArray[index], Ns, N)
    return tempArray, expectedValue

def plotDiameterStats(lowTemp, highTemp, tempStep, Ns=1500, N=30, showPlot =True):
    """
    Plot the values found in getDiameterFromTemp.
    Input:
    Same explenation as above.
    """
    tempArray, expectedValue = getDiameterFromTemp(lowTemp,highTemp, tempStep,Ns, N)
    """
    Plot the expected value and standardDeviation
    """
    if showPlot:
        plt.figure(figsize=(10,7))
        plt.plot(tempArray, expectedValue, label = r"$\langle d\rangle$")
        plt.xlabel("Temperature [K]")
        plt.ylabel("Diameter [monomerlength]")
        plt.title(r"Average Diameter as a Function of Temperature")
        plt.legend()
        plt.show()

plotDiameterStats(lowTemp=10, highTemp=1000, tempStep=30, Ns=10000, N=30)

"""
We assume that the plotting aspect of the function is comparatively fast, and we deactivate the plotting part to avoid multiple plots downwards.
```

```
"""
%timeit plotDiameterStats(lowTemp=10, highTemp=1000, tempStep=30, Ns=10000,␣
↪N=30, showPlot = False)
```




```
25.8 s ± 28.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In this project we believe that we can ultimately talk about energy instead of diameter, or vice versa, because the overall trend, and one that is quite consistent is that higher energies of the polymer accumulate to higher diameters. This is because curling up into tight spaces creates a lot of closest neighbours for the different monomers, resulting in lower energies and lower diameters. Thus, it makes sense that the expected diameter should follow the same trend as the expected energy did in task **2e)**, which seems to be the case when we compare them.

We can also see from our plot, that the diameter is consistently rising with higher temperatures, when we are in the higher temperature sections. This trend isn't necessarily valid for lower temperatures, as it could be too difficult to get out of an energy minima that isn't actually that low for a low temperature polymer.

In the previous plot, we have seen that there is quite a lot of inconsistency with the average diameter in lower temperature ranges, if we always reset the polymer. To get another understanding of the diameter at lower temperatures, we look at a new method of producing the same plot. Instead of resetting the polymer each time, we implement a cooling system, where we start at a high temperature and move our way downwards to lower temperatures. This way, it isn't that easy

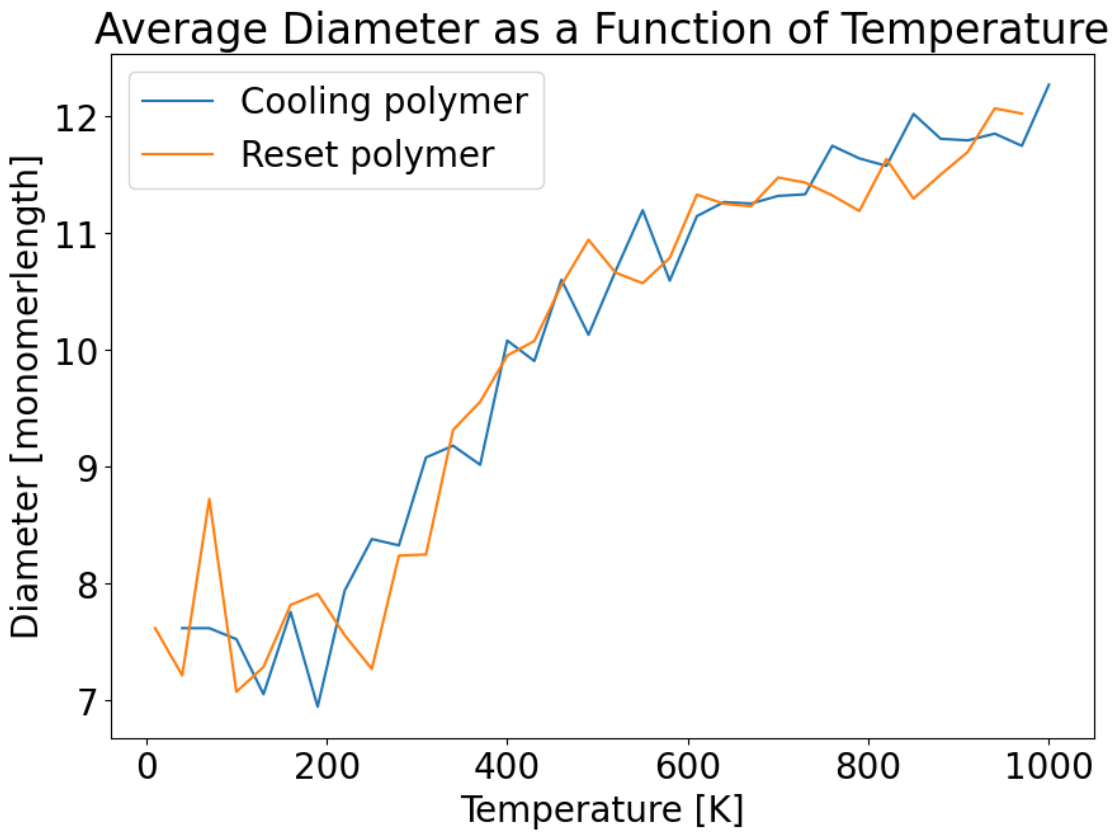
for the polymer to be stuck at different local energy minemas for different temperatures. In other words, we can predict a smoother line in the plot. The function `computeDiameterStatsV2` is quite similar to `computeDiameterStats`, but the difference is that it also takes in the polymer to simulate, so you could start with a pre-existing one, and it also returns the polymer, such that we can then access it and manipulate it further, not only look at the average diameter. The function `getDiameterFromTempV2` works very similarily to `getDiameterFromTemp`, but uses this new cooling method rather than resetting the polymer for the next temperature. The plot function `plotComparisonDiameter` plots both the version of resetting, but also the method of cooling, and is meant to shed light on the differences these two methods provide.

We have also concluded that the process where one iterates from a high temperature to a low temperature and keep the polymer configuration will simulate a cooling process.

[ ]:
```
def computeDiameterStatsV2(polymer, V, T, Ns=1500):
    """
    Identical to computeDiameterStats, just that it returns the polymer, input␣
 ↪is thus the same
    """

    polymer,_,diameter = metropolisAlgorithm(polymer, V, Ns, T,␣
 ↪includeDiameter=True)
    importantDiameter = diameter[1000:] # Note Ns > 10000

    """
    Returns the polymer and the average diameter of the polymer after the␣
 ↪simulation has stabilised (After 1000 accomplished rotations)
    """
    return polymer, np.average(importantDiameter)



def getDiameterFromTempV2(lowTemp, highTemp, tempStep, Ns, N):
    """
    Purpose, the same as getDiameterFromTemp, just where the algorithm starts␣
 ↪at the highTemp and reuse the polymer returned by each simulation.
    Input:
    Explained in 2g) in getDiameterFromTemp
    """

    polymer = createPolymer(N)  #Creating the polymer

    """
    Creating a random symmetrical matrix with values between -6*10**(-21) and␣
 ↪-2*10**(-21)
    """
    V = np.zeros((N,N))
    for i in range(N):
```

```
        for j in range(i-1):
          V[i,j]=(np.random.uniform(-6,-2))*10**(-21)

    V=V+V.transpose()


    tempArray = np.arange(highTemp,lowTemp,-tempStep) #Creating a temperature␣
 ↪array

    """
    Finding the expected diameter of the polymer at the given tempeartures
    """
    expectedValue = np.zeros(len(tempArray))
    for index in range(len(tempArray)):
        polymer, expectedValue[index]= computeDiameterStatsV2(polymer, V,␣
 ↪tempArray[index], Ns)


    return tempArray, expectedValue



def plotComparisonDiameter(lowTemp, highTemp, tempStep, Ns = 1500, N = 30):
    """
    Plot the expected diameter as a function of temperature where one simulates␣
 ↪a cooling process, and thus start at a high temperature and reuse
    the old polymer for the new simulation, while the other resets the polymer␣
 ↪for each simulation.
    Input the same as plotDiameterStats
    """
    plt.figure(figsize = (10,7))
    tempArray, expectedValue = getDiameterFromTempV2(lowTemp, highTemp,␣
 ↪tempStep, Ns, N)
    plt.plot(tempArray, expectedValue, label = "Cooling polymer")

    """
    Plot of the expected diameter as a function of temperature where the␣
 ↪polymer is reset to the default position for each simulation
    """
    tempArrayReset, expectedValueReset = getDiameterFromTemp(lowTemp, highTemp,␣
 ↪tempStep, Ns, N)
    plt.plot(tempArrayReset, expectedValueReset, label = "Reset polymer")

    """
    Fixing the appearance of the plot
    """
```

```
    plt.xlabel("Temperature [K]")
    plt.ylabel("Diameter [monomerlength]")
    plt.title(r"Average Diameter as a Function of Temperature")
    plt.legend()
    plt.show()

plotComparisonDiameter(lowTemp=10, highTemp=1000, tempStep=30, Ns=3000, N=30)
```



In this plot we mostly observe that the method of cooling polymers leave a much smoother diameter-curve, and therefore predictibly a much smoother energy-curve. We also observe as a general trend that for each simulation, the cooling curve is generally more stable, compared to the reset-curve. This is mostly highlighted by the fact that at lower temperatures, the polymer that resets changes between a wider range of possible diameters for different simulations at the same temperature, while the cooling polymer generally find the same energy minima. This makes sense, as the reset polymer only experience N_s rotations each time before it resets, while the cooling polymer iterate through all the higher temperatures before reaching the lower temperatures. Thus the cooling polymer will be rotated more times, and thus have a greater probability of ending in lower energy minimas and thus smaller diameter. It also contributes that the lower temperatures at the reset polymer rarely will be able to do a transtition that increases the diameter, as it will increase energy, which is unlikely at lower temperatures, while the cooling polymer will have gone through a stage where it was able to do those transition, and thus be more likely to consistently end up with smaller

diameter. However, it is important to keep in mind that there is a lot of inconsistencies at lower temperatures, and that the truth is that different polymers at low temperatures are quite stable at different diameters, and different energies, because the all belong to local minemas.

1.0.3 Conclusion

In conclusion, by simulating polymer rotations it is shown that the energy, temperature and diameter of a polymer are closely linked. By increasing the temperature of a polymer, both the energy and the diameter increases as well. By plotting the diameter and the energy of a polymer at comparative conditions, we've seen that said curves are very similar looking. This gives a nice relation between diameter and energy, which is useful as diameter is generally easier to measure than energy, while energy might be the characteristic one needs. Furthermore, energy for polymers at high temperatures tends to oscillate, due to the strong thermal fluctuations present at higher temperatures, whereas the energy for polymers at lower temperatures are more stable as it is more difficult to break out of an energy minima.

[ ]:
```
print(f"The entire time for the code to run is: {time.time()-startTime}")
```

```
The entire time for the code to run is: 396.49870014190674
```