Study of molecular processes through calculation of multidimensional free energy landscapes

Inauguraldissertation

zur
Erlangung der Wuerde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultaet
der Universitaet Basel

von

Wojciech Wojtas-Niziurski

aus Polen

Basel, 2016

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultaet
auf Antrag von

Torsten Schwede, Simon Bernèche and Matteo Dal Peraro

Members of the dissertation committee: Faculty representative, dissertation supervisor, and co-examiner

Basel, 24.06.2014

Date of approval by the Faculty

Prof. Dr. Torsten Schwede

_____

Signature of the Faculty representative

Prof. Dr. Joerg Schibler

_____

The Dean of Faculty

Contents

Chapter 1

Theoretical Background

1.1 Potential of Mean Force Calculations

The **Potential of Mean Force** (PMF) is one of the key concepts in modern statistical mechanics. It plays an extremely important role in the analysis of complex molecular systems. It is defined as a function of an average distribution function $\langle p(x) \rangle$:

$$\text{PMF}(x) = \text{PMF}(x^*) - k_B T \ln \left[\frac{\langle p(x) \rangle}{\langle p(x^*) \rangle} \right] \tag{1.1}$$

where x^* and PMF(x^*) are arbitrary. The average distribution function is obtained from Boltzmann average:

$$\langle p(x) \rangle \stackrel{\text{def}}{=} \frac{\int e^{-U(R)/k_B T} \delta \left(x'(R) - x \right) dR}{\int e^{-U(R)/k_B T} dR} \tag{1.2}$$

$U(R)$ is a total energy of a system as a function of all coordinates R, $x'(R)$ can represent any property of the system, or a certain combination of system's properties. The $x'(R)$ variables (which are functions of all of the degrees of freedom of a system) will be from now on called the **reaction coordinates**.

Direct calculations of a PMF from a distribution function $\langle p(x) \rangle$ is not possible in the majority of cases. It becomes clear after rewriting equation 1.2 in the form of a time average (the average values over time of the physical quantities that characterize a sys-

tem are postulated to be equal to the statistical average values of the quantities, *ergodic hypothesis*).

$$\langle p(x) \rangle = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau \delta \left(x'(t) - x \right) dt \tag{1.3}$$

Unfortunately, performing even extremely long simulation runs might not be enough to sample an entire conformational space. Usually, the existence of high energy regions on a free energy landscape can result in simulations getting stuck in a certain subset X of a conformational space $\mathcal{R}_{space}$, leaving the complementary locations $\mathcal{R}_{space} \setminus X$ unexplored. To overcome this issue many approaches have been proposed. One of those is the **Umbrella Sampling** technique. In this method, a system is simulated in presence of an artificial biasing potential $w(x)$ that restraints sampling to a specific region of a reaction coordinate space (it is usually beneficial to utilize for that function a simple harmonic function, $w(x) = \frac{1}{2} k \left(x - x_0 \right)^2$). A single biased simulation generates data narrowed down only to a small subset of a reaction coordinate space with data corresponding to a $U(R) + w(x)$ potential function. In order to acquire data from all regions of the reaction coordinate space that is functionally relevant, multiple biased simulations have to be performed with the biasing potentials defined at positions corresponding to regions of interest. Results of those simulations have to be unbiased and recombined together to produce an estimate of PMF(x). Taking equation 1.2 and substituting a potential function with a biased window potential one produces a biased simulation distribution function:

$$\langle p(x) \rangle_{(i)} = \frac{\int e^{-U(R) - w_i(x'(R)) \big/ k_B T} \delta \left(x'(R) - x \right) dR}{\int e^{-U(R) - w_i(x'(R)) \big/ k_B T} dR} \tag{1.4}$$

$$\langle p(x) \rangle_{(i)} = \frac{\int e^{-U(R) \big/ k_B T} dR}{\int e^{-U(R) \big/ k_B T} dR} \times \frac{\int e^{-U(R) \big/ k_B T} e^{-w_i(x'(R)) \big/ k_B T} \delta \left(x'(R) - x \right) dR}{\int e^{-U(R) \big/ k_B T} e^{-w_i(x'(R)) \big/ k_B T} dR} \tag{1.5}$$

$$\begin{aligned} \langle p(x) \rangle_{(i)} = e^{-w_i(x) \big/ k_B T} \times \frac{\int e^{-U(R) \big/ k_B T} \delta \left(x'(R) - x \right) dR}{\int e^{-U(R) \big/ k_B T} dR} \\ \times \frac{\int e^{-U(R) \big/ k_B T} dR}{\int e^{-U(R) \big/ k_B T} e^{-w_i(x'(R)) \big/ k_B T} dR} \end{aligned} \tag{1.6}$$

$$\langle p(x)\rangle_{(i)} = \frac{e^{-w_i(x)/k_BT}\,\langle p(x)\rangle}{\left\langle e^{-w_i(x)/k_BT}\right\rangle} \tag{1.7}$$

As a result, the PMF from the i-th windows is:

$$\mathrm{PMF}_{(i)}(x) = \mathrm{PMF}(x^*) - k_BT \ln\left[\frac{\langle p(x)\rangle_{(i)}}{\langle p(x^*)\rangle}\right] \tag{1.8}$$

$$\mathrm{PMF}_{(i)}(x) = \mathrm{PMF}(x^*) - k_BT \ln\left[\frac{\langle p(x)\rangle}{\langle p(x^*)\rangle}\right] - w_i(x) + F_i \tag{1.9}$$

Where the F_i constant is defined from:

$$e^{-F_i/k_BT} = \left\langle e^{-w_i(x)/k_BT}\right\rangle \tag{1.10}$$

Merging data from multiple windows comes down to determining F_i constants for all windows. This can be done by modifying F values of adjacent windows i and j, until $\mathrm{PMF}_{(i)}(x)$ and $\mathrm{PMF}_{(j)}(x)$ overlap in a region where both $\mathrm{PMF}_{(i)}(x)$ and $\mathrm{PMF}_{(j)}(x)$ are defined. This also implies that biasing potentials should be defined in such way that there should exist a path connecting points A and B belonging to simulation data of windows i and j for each pair of windows at $x_{0(i)}$ and $x_{0(j)}$; $A \in \bigcup_{0\le t\le T_i}\{x_{(i)}(t)\}$, $B \in \bigcup_{0\le t\le T_j}\{x_{(j)}(t)\}$. More on paths and energy landscape traversal in chapter 1.3.2.

1.2 Weighted Histogram Analysis Method

One of the most recognised methods for unbiasing is the Weighted Histogram Analysis Method [13]. Having performed N biased window simulations, WHAM equations represent the estimate of an unbiased density function 1.2:

$$\langle p(x)\rangle = \sum_{i=1}^{N}\left\{\langle p(x)\rangle_{(i)}^{\text{unbiased}} \times \frac{n_i \cdot e^{-[w_i(x)-F_i]/k_BT}}{\sum_{j=1}^{N} n_j \cdot e^{-[w_j(x)-F_j]/k_BT}}\right\} \tag{1.11}$$

where n_i represents the number of data points obtained from i-th windows simulation. Comibning 1.7 and 1.10, a single window simulation unbiased density can be written:

$$\langle p(x)\rangle_{(i)} = \frac{e^{-w_i(x)\big/k_BT}\,\langle p(x)\rangle^{\text{unbiased}}}{e^{-F_i\big/k_BT}} \tag{1.12}$$

$$\langle p(x)\rangle^{\text{unbiased}} = \frac{\langle p(x)\rangle_{(i)}}{e^{-[w_i(x)-F_i]\big/k_BT}} \tag{1.13}$$

With that, the 1.11 equation can by rewritten in this form:

$$\langle p(x)\rangle = \frac{\sum_{i=1}^{N} n_i\,\langle p(x)\rangle_{(i)}}{\sum_{j=1}^{N} n_j \cdot e^{-[w_j(x)-F_j]\big/k_BT}} \tag{1.14}$$

The 1.10 equation also can be rewritten as:

$$e^{-F_i\big/k_BT} = \left\langle e^{-w_i(x)\big/k_BT}\right\rangle \overset{\text{def}}{=} \int e^{-w_i(x)\big/k_BT}\,\langle p(x)\rangle\,dx \tag{1.15}$$

Equations 1.14 and 1.15 provide together an estimate of the unbiased distribution function. Based on that estimate, the final structure of a Potential of Mean Force is calculated from equation 1.1. Since 1.14 and 1.15 are codependent, they need to be solved self-consistently. This is achieved by iterating those equations until predefined criteria for parameter convergence are met. The procedure starts with a certain initial guess of $\{F_i\}$ values.

One on the biggest advantages of using the Weighted Histogram Analysis Method is its extendibility up to (theoretically) any number of dimensions N_D:

$$\langle p(x_1,\ldots,x_{N_D})\rangle = \frac{\sum_{i=1}^{N} n_i\,\langle p(x_1,\ldots,x_{N_D})\rangle_{(i)}}{\sum_{j=1}^{N} n_j \cdot e^{-\left[w_j(x_1,\ldots,x_{N_D})-F_j\right]\big/k_BT}} \tag{1.16}$$

$$e^{-F_i\big/k_BT} = \int\cdots\int e^{-w_i(x_1,\ldots,x_{N_D})\big/k_BT}\,\langle p(x_1,\ldots,x_{N_D})\rangle\,dx_1\ldots dx_{N_D} \tag{1.17}$$

1.2.1 Basic N_D-dimensional implementation

The general approach towards implementing the Weighted Histogram Analysis Method assumes having performed a N_D-dimensional Umbrella Sampling simulation with N windows. Each (i-th window) defined using harmonic biasing potential at $x_1^i \dots x_{N_D}^i$ with force constants $k_1^i \dots k_{N_D}^i$. The algorithm (algorithm 1) alone is extremely simple and can be easily optimised (chapter 2.1).

Algorithm 1: Weighted Histogram Analysis Method

input : H; // biased data
output: R; // unbiased data
output: PMF; // potential of mean force

while ***not*** *converged* **do**
  for $i = 0, i < N, inc(i)$ **do**
    $F_i' \leftarrow 0$
  end
  foreach *position in H* ***as*** $x_1 \dots x_{N_D}$ **do**
    $d \leftarrow 0$
    for $i = 0, i < N, inc(i)$ **do**
      $d \leftarrow d + n_i \cdot e^{F_i / k_B T} \; e^{-\left\{ \frac{1}{2} k^i{}_1 \left(x_1 - x^i{}_1\right)^2 + \dots + \frac{1}{2} k^i{}_{N_D} \left(x_{N_D} - x^i{}_{N_D}\right)^2 \right\} / k_B T}$
    end
    $\mathrm{R}[x_1 \dots x_{N_D}] \leftarrow \mathrm{H}[x_1 \dots x_{N_D}] \; / \; d$
    for $i = 0, i < N, inc(i)$ **do**
      $F_i' \leftarrow F_i' + \mathrm{R}[x_1 \dots x_{N_D}] \cdot e^{-\left\{ \frac{1}{2} k^i{}_1 \left(x_1 - x^i{}_1\right)^2 + \dots + \frac{1}{2} k^i{}_{N_D} \left(x_{N_D} - x^i{}_{N_D}\right)^2 \right\} / k_B T}$
    end
  end
  for $i = 0, i < N, inc(i)$ **do**
    $F_i' \leftarrow -k_B T \cdot \log F_i'$
    check (abs (F_i - F_i'))
    $F_i \leftarrow F_i'$
  end
end
foreach *position in H* ***as*** $x_1 \dots x_{N_D}$ **do**
  $\mathrm{PMF}[x_1 \dots x_{N_D}] = -k_B T \cdot \log \left(\mathrm{R}[x_1 \dots x_{N_D}] / \max(\mathrm{R}[\dots]) \right)$
end

able free energy valleys and to avoid high (steep) energy barriers (in many cases going over those high energy regions is not beneficial at all, forcing a system in an energy unfavorable conformation with artificial biasing potentials might result in system undergoing unphysiological changes). Consequently, these observations could be utilized to narrow down an area of interest to a subset of the reaction coordinate space that would correspond to energies that are not exceeding a certain value E_0.

$$\mathrm{E}\left(i_1, i_2, \ldots, i_{N_D}\right) = \mathrm{PMF}\left(\left(\begin{array}{c} x_{01} \\ \vdots \\ x_{0N_D} \end{array}\right)^T + \left(\begin{array}{c} \Delta x_1 \\ \vdots \\ \Delta x_{N_D} \end{array}\right)^T \cdot \mathrm{diag}\left(i_1, i_2, \ldots, i_{N_D}\right)\right) \leq E_0$$

This relation is true only for a certain subset of the whole space, therefore it could possibly be unnecessary to store an energy value for every single position in matrix $(i_1, i_2, \ldots, i_{N_D}) \in \{0, \ldots, n_0\} \times \ldots \times \{0, \ldots, n_{N_D}\}$. Instead, all of the indexes for which energy is defined could be stored in a certain set, let's call it X.

$$(i_1, i_2, \ldots, i_{N_D}) \in X \Leftrightarrow \mathrm{E}\left(i_1, i_2, \ldots, i_{N_D}\right) \leq E_0$$

If the number of elements in the set X is not bigger than a half of number of elements in multidimensional array

$$2 \cdot \mathrm{card}\left(X\right) \leq \mathrm{card}\left(\{0, \ldots, n_0\} \times \ldots \times \{0, \ldots, n_{N_D}\}\right)$$

then the **matrix** representing an energy landscape is **sparse**.

Building a data structure for storing energy landscapes on top of a certain **sparse matrices** implementation could result in significant reduction of memory usage. In fact, in a simple 2D test simulation, the size of a full energy map was 262 KB against 69 KB for a sparse matrix stored with a cutoff of $E_0 = 10$ kcal/mol. For a sparse matrices implementation I've decided to use a **dictionary of keys**. The information about positions for

which data is defined would be stored in a self-balancing binary search tree (BST), and the basic procedures (insertion, removal, lookup) would be performed in $O(\log n)$ time and the extraction of all stored items would take only $O(n)$ time [1,11]. In C++ a sparse matrix can be easily implemented as a standard template library map, with basic function defined as follows:

Listing 1.1: sparse matrix, data structure creation

```
data = new std::map < int * , T > ();
```

Listing 1.2: sparse matrix, adding/modifying an element

```
data->operator [] (position) = value;
```

Listing 1.3: sparse matrix, accessing an element

```
value = data->operator [] (position);
```

Besides obvious advantages, there are certain downsides to the use of sparse matrices for an array implementation. The major one is the time for accessing values at the neighboring locations; $O(\log n)$ for each query. However application of this data structure makes storing of the huge (multidimensional) data sets possible, and that compensates for the fact of the calculation time not being optimal. The important fact is that utilizing such data structure does not affect the performance of WHAM procedure (chapter 1.2). In a matter of fact, certain modifications allow simplifications in the main loop of WHAM algorithm that significantly improve the calculation time. This matter will be discussed later (chapter 2.1).

1.3.2 Graph Representation

Another data structure which can be used to represent energy landscapes is a **graph**. Let $G = (V, N)$ be an **undirected connected graph** representing an energy function E with discretized parameters (V is a set of nodes, N is a set of edges).

Lets define a node $v \in V$ as a object defined by an absolute position in a reaction coordinate space and a free energy value (listing 1.4).

Listing 1.4: Node class definition

```
template <typename T, typename U>
class node
{
private:
    int index                           // unique index
    std::vector<int> connections;       // indexes of adjacent nodes

public:
    T * coordinates;                    // position
    U value;                            // free energy value
};
```

In my implementation (listing 1.5), which served as a valid solution for certain analyses I have been performing in my research, a graph class contains a list (a vector) of all nodes and a binary search tree (a map) that is utilized as a dictionary for accessing a neighborhood map (removing a need for its explicit implementation). These implementation choices result in a $O(\log n)$ time for performing queries and accessing nodes at specific locations, $O(n \log n)$ time for a complete construction of a graph with n nodes, and a $O(1)$ time for accessing neighbors of a node. The last statement might not be obvious, therefore it can be beneficial to perform slightly deeper analysis of those procedures:

- When a new node n is added to a graph:

  A list and a map get a new entries pointing to n (dictionary uses a position of n as a key for indexing elements),

  A procedure responsible for connecting nodes with each other is executed.

  Adding an element to a list is performed in $O(1)$ time; adding elements to a map, as well as finding adjacent nodes takes $O(\log n)$ (since a map is a balanced BST). Hence, the whole process of adding a single node to a graph takes $O(\log n)$ time.

- Therefore the construction of a whole graph (adding n elements one after another) is executed in $O(n \log n)$ time.

- In a graph created with this procedure, every single node contains an information about nodes adjacent to it. This information is implemented as a vector of foreign indexes. Those indexes point to certain positions in a list of nodes in a graph object and are used for accessing elements in constant time. Since only lists are involved in this procedure, the whole function is executed in $O(1)$ time.

Listing 1.5: A simplified graph class

```
template <typename T, typename U>
class graph
{
private:
    std::vector < node<T,U> * >          nodes;
    std::map    < T * , node<T,U> * >    lookup;
    void connect ( node<T,U> * );

public:
    graph();
    ~graph();

    void addNode ( node<T,U> * N ) {
        this->nodes.push_bach (N);
        this->lookup.operator[] (N->coordinates) = N;
        this->connect (N);
    };
};
```

Connected Graphs

In order to have WHAM functions converging properly it is important for the reaction coordinate space to be sampled continuously (definition 1).

Definition 1.

Let G be a graph representing an energy function E.
*E is **continuous** if and only if G is **connected** (definition 2).*

Definition 2.

*A graph is **connected** if there is a path from any point to any other point in the graph.*

Problem of determining whether a graph is connected (definition 2) or disconnected [1] in a undirected graph can by solved by running **Breadth-first Search** (BFS, algorithm 2) starting at any node $v \in G$ and checking if all nodes in a graph have been visited by the algorithm.

Algorithm 2: Breadth-first Search

```
input  : G;                                   // an initial graph
input  : n;                                   // a starting node
output: V;                                    // a set of all visited nodes
1  Q ← queue()
2  V ← set()
3  Q.enqueue(n)
4  V.add(n)
5  while Q not empty do
6      t ← Q.deqeue()
7      foreach node e adjacent to t do
8          if e not in V then
9              Q.enqueue(e)
10             V.add(e)
11         end
12     end
13 end
```

[1] A disconnected graph is a graph which is not connected.

Chapter 2

Methods and Implementaions

Before we can move from theory to running real potential of mean force calculations, certain structures defined in the previous chapter (chapter 1) require a more in-depth analysis. In this chapter I will go over certain design/implementation principles I have chosen for the application.

A big part of my project was to build an application that could be easily used for running multidimensional PMF calculations (chapter 4). The application itself uses an umbrella sampling based approach for calculating potentials of mean force (chapter 1.1), and executes weighted histogram analysis method (chapter 1.2) to remove the effects that biasing potentials have on the data. In chapter 2.1 I will perform a brief performance analysis of WHAM approach, I will discuss advantages and disadvantages of implementing certain optimizations, and at the end I will present a shape of what seems to be the optimal WHAM implementation (chapter 2.1.4). After that I will move to the description of the data model that was implemented (chapter 2.2). It is clear that handling of big data sets from simulations ran in multidimensional spaces might require some special attention, therefore it is extremely important to understand how this process can be tweaked to allow optimal running times, and to reduce the effect the data handling process has on other processes running on a computational clusters.

Contents

2.1 Optimising WHAM

The procedure of unbiasing data from a N_D-dimensional Umbrella Sampling calculations performed with N window simulations can be accomplished with the WHAM algorithm (chapter 1.2, algorithm 1). Application of this procedure to some large input data sets can be very slow. In each cycle of the main loop (algorithm 1, line 1) numerical calculations are performed for each position (line 5) for each window (lines 7 and 11). Therefore, in each cycle of the main loop $C \cdot N \cdot \Pi_{i \in \{0,\ldots,N_D-1\}} n_i$ operations are performed, C is a constant representing the amount of floating point operation in the inner-most loops of WHAM. The total CPU cost of the whole procedure is even higher[1].

2.1.1 Reducing number of operations per cycle

The first thing that can be tweaked in the implementation is a reduction of the C constant. It is clear that the big exponent in lines 8 and 12 (algorithm 1) is *a*) equal; *b*) requires quite

[1] Number of CPU operations for different math operations:

- 1 operation for additions/subtractions;
- 4 operations for multiplications;
- 10 operations for division;
- 50 operations for exponents and logarithms

a few operations to be calculated: $4N_D + 1$ multiplications[2] , $2N_D - 1$ additions[3], 1 negation[4] and 1 exponent (which adds up to $50 + 18N_D + 4$ operations on a CPU); *c*) doesn't change over the algorithms main loop cycles. Occurring twice in the algorithm and multiplied by a number of windows, the total number of operations on a single CPU adds up to $108N + 36N_DN \geq 144N$. The remaining part of the main loop consists of $5N$ multiplications, $2N$ addictions, N divisions, N exponents, N logarithms and N assignments with a total sum of $5N \cdot 4 + 2N \cdot 1 + N \cdot 10 + N \cdot 50 + N \cdot 50 + N \cdot 1 = 133N$ operations on a CPU. Pre-calcluating the exponent from lines 8 and 12 would therefore result in a speed increase of 108% in 1D, 162% in 3D, and 243% in 6D. These numbers are quite big, however this particular optimization might not be realistic in typical implementations. As mentioned before (chapter 1.3, Data Structures), the major issue with multidimensional PMF calculations is storage, and pre-calculating exponents from lines 8 and 12 would require additional $8 \cdot N \cdot \Pi_{i \in \{0,\dots,N_D\}} n_i$ bytes of storage[5] which is not realistic in a typical WHAM implementation (algorithm 1). Application of this optimizations becomes possible only after merging it together with some additional ones.

2.1.2 Application of Sparse Matrices

As described in chapter 1.3.1, a **sparse matrix** based data structures can be used to represent multidimensional arrays. The real size of data stored like that might be significantly smaller then if a full array was used. Hence, the number of positions (listing 2.1) for which explicit calculations are performed in the main loop of WHAM procedure (algorithm 1) is drastically reduced.

Listing 2.1: Iterating through the positions in an array representing the energy landscape

```
foreach position in H as x_1 ... x_{N_D}
```

[2]Division by k_BT is in fact multiplication by a pre-calculated constant $1/k_BT$.

[3]Additions count includes also subtractions .

[4]Negation is as expensive computationally as subtraction ($-x = 0 - x$).

[5]In a 3D Umbrella Sampling simulation with 1000 windows a data resolution of $n_0 \times n_1 \times n_2$, $n_0 = n_1 = n_2 = 1024$, the total amount of memory required to store the pre-calculated exponents would be around 60GB.

Utilization of **sparse matrices** in combination with pre-calculation of exponents (chapter 2.1.1) might be enough to allow a WHAM implementation to run within available RAM[6].

2.1.3 Removing Dimensional Dependence

In any computational problem, if possible, it is extremely beneficial to get rid of concept of dimensions. In WHAM (algorithm 1) only one loop is dimension-dependent. To get a better understanding of what exactly is happening there, let's have a closer look at an explicit 3D implementation of this loop (listing 2.1).

Listing 2.2: Iterating through the positions in a 3D array.

```
for (int i = 0; i < x_0; i++)
        for (int j = 0; j < x_1; j++)
                for (int k = 0; k < x_2; k++) {
                        // perform calculations
                }
```

It might not be obvious that running N_D nested loops in every cycle of the main loop in WHAM algorithm could slow the whole procedure down. The key to realizing this fact is understating how does the data structure for storing energy landscapes work. Iterating through the elements of a multidimensional array might require a lot of "jumping" within reserved blocks of memory (such operations are not optimal due to increased times for data access). The question is, if it is possible to replace the N_D nested loops with a single one that would sequentially go through all of the elements. This can be achieved by converting a multidimensional array to a vector (listing 2.3) before moving forward to running the main loop of WHAM. In this concept a N_D-dimensional data structure is represented by two vectors: one that contains values (listing 2.3, line 1), and second that stores the positions from a "real" array (listing 2.3, line 2).

[6] Running WHAM, implemented with Sparse Matrices and exponents pre-calculation, for a 3D input data set the required memory may get above 1GB.

The next obvious step here is to replace an array representation of an input data set with a sparse matrix (chapter 2.1.2). This is the key to having an optimal WHAM implementation. Storing data in a sparse matrix as a dictionary of keys (associative array, binary search tree [1,7,11]) provides a very fast and elegant interface for accessing simulations data (chapter 1.3.1). Data pre-processing procedure updated with sparse matrices is presented in listing 2.4.

Listing 2.3: Pre-processing data to avoid dimensional-dependence in the main loop of WHAM. Variable `H` is a biased histogram, and function `position(i,j)` represents a discretised position at j along i-th dimension.

```
vector <double> histogram;       // pre-processed biased data
vector <double*> positions;      // positions for histogram variable

for (int i=0; i<x_0; i++)
        for (int j=0; j<x_1; j++)
                for (int k=0; k<x_2; k++) {
                        double * pos = new double[3];
                        pos[0] = position (0,i);
                        pos[1] = position (1,j);
                        pos[2] = position (2,k);

                        positions.push_back (pos);
                        histogram.push_back (H(i,j,k));
                }
```

Listing 2.4: Pre-processing data to avoid dimensional-dependence in the main loop of WHAM (implemented with sparse matrices). Variable `data` represents an input unbiased histogram and is of type `std::map < double *, double >`.

```
for (int i = data.begin(); i != data.end(); i++) {
        double * pos = new double[3];
        pos = i->first;
        positions.push_back (pos);
        histogram.push_back (i->second);
}
```

With data pre-processed like that (listings 2.3, 2.4), the loop responsible for iterating through the elements in a multidimensional space becomes a single loop through the elements of a 1D array (listing 2.5). It is also worth mentioning one more time that sparse matrices implemented with associative arrays use binary search trees as a low level implementation [1,7,11], as a result the time required to pre-process biased data is optimal (linear, $O(n)$).

Listing 2.5: Intreating through the positions in an pre-processed array.

```
for (int i=0; i<histogram.size(); i++) {
        position = positions[i];          // x_1 ... x_{N_D}
        value = histogram[i];             // H[x_1 ... x_{N_D}]

        // perform calculations
}
```

Incorporating this approach is extremely beneficial since:

- It cleans up the code, and makes it more readable without having to pay any price in performance.
- The memory manager doesn't have to jump all over the reserved blocks of memory, instead data is read sequentially (which results in significantly lower access times).

- One code with a number of dimensions (N_D) as a parameter can handle the unbiasing of data in theoretically any number of dimensions.

2.1.4 Parallelization

WHAM calculations can get quite expensive computationally, especially in multidimensional spaces. Therefore it is clear that it would be worth investigating on the possibility of parallelizing the algorithm.

Fact 1. *WHAM algorithm is parallelizable.*

This might not be obvious when looking at the nested loops in the core loop of WHAM procedure (listing 2.2), however after the conversion of a multidimensional array to a vector (listing 2.4), the calculations are performed sequentially (listing 2.5). It is also extremely important to state the fact that all of those simulations are independent, which implies that each of those could be executed at a different processing unit (algorithm 1, listing 2.5).

Fact 2. *Let **n** be a number of elements in a vector representing a multidimensional array used for storing biased simulation data, and let **m** be a number of available processing units. Time complexity of the main loop in WHAM algorithm is **O(n/m)**.*[7]

Parallelization of WHAM simply requires that the calculations be distributed by splitting a vector containing sampling data among the processing units (listing 2.6). The very same code is executed simultaneously on all processors, and the data exchange between them is managed by the Message Passing Interface (MPI). Each CPU performs calculations on a list of coordinates restricted to a certain range in a vector representing data (listing 2.7). Synchronization of values between CPUs is performed after each cycle of the main algorithm's loop in order to update F_i values (chapter 1.2, listing 2.8).
MPI version of WHAM implemented to utilize sparse matrices for storing data is one of the most optimal implementations possible. In one 3D simulation test, using 32 CPUs to unbias data and to calculate a potential of mean force took around 3 minutes, in comparison to over 1.5h on a single CPUw.

[7] Increasing the number of CPUs k-times results in k-times faster WHAM runs.

There are other interesting ways to improve WHAM's performance. One worth mentioning is constructing a script that would generate and compile a C++ code to calculate a free energy landscape specifically for a given problem. This is a valid approach, since we can use static arrays with a fixed size which offer far better performance then dynamically allocated arrays. I decided not to publish my code representing this procedure, since due to its level of complication it is not easily readable.

Listing 2.6: Determining how a vector containing data is going to be split among the available processing units.

```
int PE_size     = MPI::COMM_WORLD.Get_size ();  // number of CPUs
int PE_id       = MPI::COMM_WORLD.Get_rank ();  // current CPU

// number of positions in a vector assigned to a sequence of CPUs
int * mpi_data          = new int [PE_size];
int * mpi_data_cells    = new int [PE_size+1];

// number of positions in a vector per CPU
int cut = dS / PE_size;
for (int i = 0; i < PE_size; i++)
        mpi_data[i] = cut;

// fixing the data size in compliance with divisions reminder
for (int i = 0; i < dS-cut*PE_size; i++)
        mpi_data[i]++;

// determining data range for each CPU
mpi_data_cells [0] = 0;
for (int i = 0; i < PE_size; i++)
        mpi_data_cells[i+1] = mpi_data_cells[i] + mpi_data[i];
```

Listing 2.7: Structure of the main loop in WHAM implemented with MPI. Each CPU runs the main loop, however calculations for all of them are restricted to a certain range in a vector representing unbiased simulation data.

```
while ((iCycle < nStepMAX) && (Cycle)) {
        for (int i = 0; i < nW; i++)
                Fb[i] = 0.0E0;
        // looping through positions depends on an id of a CPU
        for (int k=mpi_data_cells[PE_id]; k<mpi_data_cells[PE_id+1]; k++) {
```

Listing 2.8: Merging values of the F list. All CPUs send their F_i values to the first CPU, and their further analysis is performed only there. After that's done, values will be broadcasted back by the first CPU.

```
// synchronization of F values
if (PE_id != 0)
        MPI::COMM_WORLD.Send ( Fb , nW , MPI::DOUBLE, 0, 1 );
else {
        for (int i = 1; i < PE_size; i++) {
                MPI::COMM_WORLD.Recv ( FF , nW , MPI::DOUBLE, i, 1);
                for (int j = 0; j < nW; j++)
                        Fb[j] += FF[j];
        };
};
// 1st CPU performs convergence check
if (PE_id == 0) {
        Cycle = false;
        for (int i = 0; i < nW; i++) {
                Fb[i] = -kBT * log ( Fb [i] );
                diff = fabs ( Fb[i] - Fb[0] - F[i] + F[0] );

                if (diff > tolerance)
                        Cycle = true;
        };

        for (int i = 0; i < nW; i++)
                F[i] = Fb[i];
};
// 1st CPU broadcasts F values to all other CPUs
// along with an instruction if the main loop should continue running
MPI::COMM_WORLD.Barrier();
MPI::COMM_WORLD.Bcast( F , nW , MPI::DOUBLE , 0);
MPI::COMM_WORLD.Bcast( &Cycle , 1 , MPI::BOOL , 0);
```

2.2 Data input/output

Handling big data sets is not an easy task. Let's consider a situation, where WHAM algorithm (chapter 1.2) is used to calculate a PMF from an Umbrella Sampling simulations with 500 windows. Total time of each window simulation is 1 ns, with reaction coordinates values being stored every 2 fs in 1 ps segments (one data file written every 1ps). This adds up to 1000 files and $5 \cdot 10^5$ data points for a single window, and $5 \cdot 10^5$ files and $25 \cdot 10^7$ data points for the whole simulation. Assuming that only the second half of each simulation represents a converged system (is significant for the free energy calculation), a number of files that has to be read in order to build the unbiased sampling map is $2.5 \cdot 10^5$. This typically is not even an issue, however it might become an extremely serious problem in computer environments with multiple read/write operations, for example on clusters where disc drives are shared among multiple users.

The first thing that can be implemented in order to reduce a number of files is data compression. Each window would be associated with a single archive containing all the data it produced. Using archives indeed simplifies the data management process significantly, instead of loading 500 files into memory only one file is being loaded. Reduction of size by compression is also an important factor, however its impact on the overall performance can be ignored.

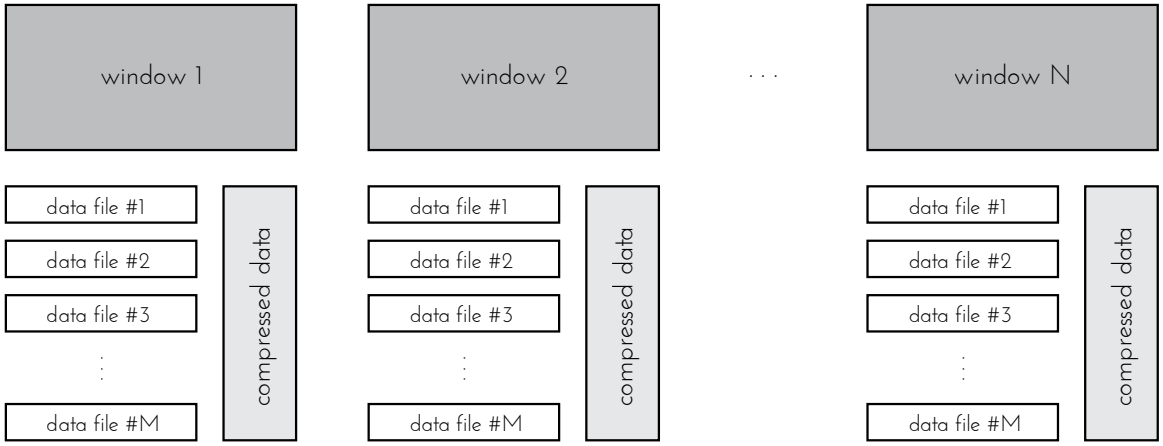


Figure 2.1: Basic windows data structure

Another thing that can be done is completely rethinking how the data is stored. Instead of explicitly saving the data, only a particular data representation can be saved. For various reasons that have already been mentioned in chapter 1.3, Sparse matrices were used as a data structure for storying free energy maps in memory. The explicit implementation also includes a dedicated file format for storing sparse data:

- First line of this file defines data type (listing 2.9, line 1). Two options are available:
  - `# sparse.histogram`
  - `# sparse.data`
- Second line contains two numbers (listing 2.9, line 2):
  1. Number of dimension N_D of keys in a sparse matrix
  2. Value of each positions in a sparse matrix that is not defined explicitly. Traditionally a sparse matrix definition assumes that most elements in the matrix are equal to 0, here I am extending the definition so that most of elements in the matrix are equal to certain $X_S \in \mathbb{R}$.
- Next N_D lines determine the dimensions of a matrix (listing 2.9, lines 3-5). Each of those lines consists of three values:
  1. Lower bound of a reaction coordinate
  2. Upper bound of a reaction coordinate
  3. A step in a discretized representation of reaction coordinates
- All lines after that define the actual structure of the map. Every lines consists of N_D+1 values (listing 2.9, lines 6-10):
  - First N_D values point to a position where the values is defined
  - The last position in a row is a value

Listing 2.9: First 10 lines of a sparse data file.

```
# sparse.data
# 3 23.7373
# -15.75  5.75 0.1
#  -9.75  9.75 0.1
#  -7.75 15.75 0.1
-12.75  -3.95   1.15    7.12110
-12.75  -3.85   1.15    6.51529
-12.75  -3.85   1.25    6.18277
-12.75  -3.85   1.35    5.60430
-12.75  -3.85   1.45    5.57825
```

Basic operations, like reading (definition: listing 2.10; implementation: appendix, listing **??**) and saving (definition: listing 2.11; implementation: appendix, listing **??**) are implemented in C++ and exported to Python via boost package.

Listing 2.10: Saving data as a sparse data file.

```
1 template < typename T, typename U >
2 void saveSparseData ( boost::shared_ptr < dataGrid <T,U> > , std::string );
```

Listing 2.11: Loading data from a sparse data file.

```
1 template < typename T, typename U >
2 void loadSparseData ( boost::shared_ptr < dataGrid <T,U> >, std::string );
```

This data model can be utilized to store data differently. The idea is that the regular data representation[8] could be replaced[9] with sparse matrices $S_i^{(j)}$ defined for each iteration i of every window j. $S_i^{(j)}$ is defined as a data container with the information about all unbiased data from j first iteration of window i. Each sparse matrix is build based on the

[8] Regular data representations is: one data file for each cycle of each window.

[9] For safety, both representations are stored, however the explicit one is used only as a backup.

sparse matrices from the previous iteration.

$$S_i^{(j)} = S_{i-1}^{(j)} + \text{SparseHistogram}\left(\text{data}_i^{(j)}\right) \tag{2.1}$$

Application of this data implementation significantly improves the process of retrieving simulation data. Unbiased sampling from the last 500ps of simulation[10] is equal to:

$$\sum_{n=250}^{500} \text{data}_n^{(j)} = S_{500}^{(j)} - S_{250}^{(j)} \tag{2.2}$$

Instead of loading 250 files with 500 lines each, only two files with < 200 lines are read. This data handling method reduces data retrieval time by a factor of over 300.

This final data model for storying unbiased simulation data is included by default in the iPMF application (figure 2.2).

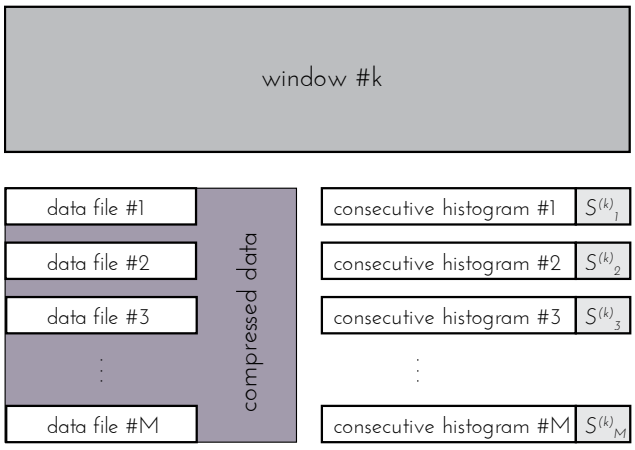


Figure 2.2: Full single window data structure

[10] Assuming that each window simulation is 1ns; 500 cycles of 2ps.

Chapter 3

The Self-Learning Method

3.1 The Self-Learning Adaptive Scheme

The aim of any PMF calculation approach should be to describe the free energy landscape within a subspace of predefined reaction coordinates with the greatest accuracy and a minimal sampling effort. Stratified US is arguably the most accurate approach to this task, but it can be computationally expensive in high dimensionality. This limitation can be circumvented if sampling via computationally costly simulations is limited to regions of the subspace of collective variables where the PMF is below a certain maximum threshold. To achieve this, the self-learning adaptive umbrella sampling process progressively builds simulation windows at positions indicated by the ongoing sampling data.

Like for any stratified US approach, an appropriate list of N_D reaction coordinates x_i with their respective boundaries needs to be determined. A biasing potential, usually defined as $w(x) = \frac{1}{2}k_i \left(x - x_0\right)^2$, and an interval for window creation, Δx_i, are also required for each reaction coordinate. In our current implementation, k_i and Δx_i are fixed, but this is not a requirement of the approach. The sampling could be made even more efficient by adjusting on-the-fly these values to the local features of the free energy landscape. This is made possible by the flexibility of the WHAM algorithm (chapter 1.2) that is used to combine the sampling data provided by the different windows.

The process starts with a system located somewhere within the N_D-dimensional reaction coordinate space. The creation of a minimal number of simulation windows is required to perform a first assessment of the local free energy landscape. In this first step, $3N_D$ windows are created. The position $(x_1, \ldots, x_{N_D})$ of each initial simulation window is determined by taking the starting state of the system $\left(x_1^A, \ldots, x_{N_D}^A\right)$ and changing its reaction coordinates x_i^A by small values Δx_i in both directions, i.e.

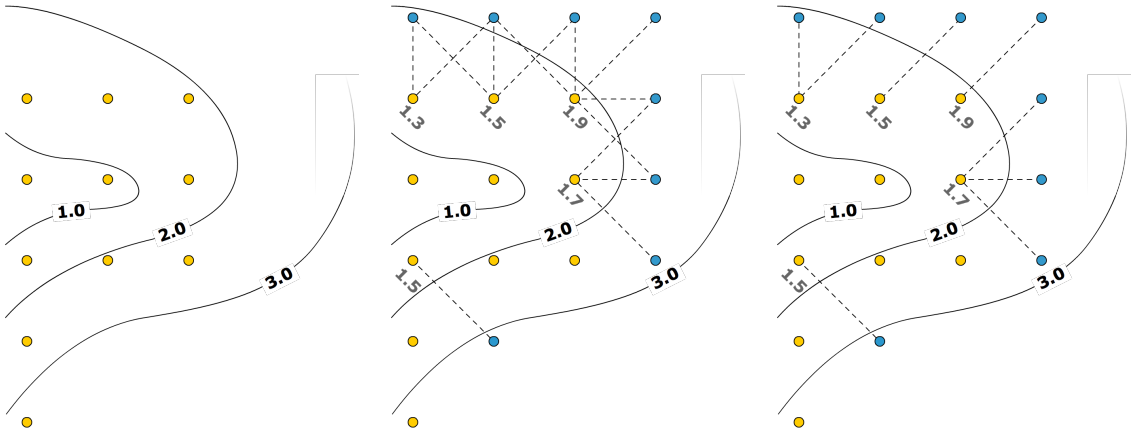
$$(x_1, \ldots, x_{N_D}) \in \Pi_i \left\{x_i^A - \Delta x_i, x_i^A, x_i^A + \Delta x_i\right\}$$

The PMF exploration is initialized on the basis of these 3^{N_D} windows from which a first PMF is calculated using the WHAM algorithm. The procedure responsible for creating new windows in regions that remain to be explored is based on the current view of the free energy landscape, at the periphery of which new windows are constructed. A parameter W_{max} is introduced in order to guide the exploration of the subspace X: no new windows can be created in the areas of the reaction coordinate space where the free energy is higher than W_{max}. To favor the exploration of lower free energy pathways while allowing exploration of pathways with higher free energy barriers when needed, W_{max} is initially set to a low value (e.g., 2 kcal/mol) and is incrementally increased up to a predefined limit (e.g., 10 kcal/mol) if the algorithm fails to create new windows at a given cycle. The procedure can be summarized as follows (see Figure 3.1).

1. The free energy landscape is calculated using the WHAM algorithm once all windows have provided a certain minimal amount of sampling data. (Figure 3.1a illustrates such a free energy landscape calculated from 11 windows.)

2. Among all existing windows, those with a free energy value lower than W_{max} (initially set to W_{max} = E_1) are selected as a base for the expansion procedure. (In Figure 3.1b, W_{max} = 2.0, and thus seven windows could potentially be used for expansion.)

3. Each of the preselected windows attempts to create a new window in $3N_D - 1$ neighboring locations. Locations already occupied by windows are omitted. (In Figure 3.1b, five windows are selected to create eight new windows.).



(a) (b) (c)

Figure 3.1: Free energy landscape exploration procedure. (a) The yellow dots correspond to the positions of 11 windows plotted on top of a free energy landscape generated with sampling from these windows. (b) Windows that are located in regions with a free energy below a given level W_{max} (here $W_{max} = 2.0$) attempt to create new windows on free neighboring grid points. (c) When more than one window targets the same location, the window associated with the lowest free energy is selected as the source of the system conformation to initiate the new window.

4. In the case when two or more windows want to expand to the same location, the window with the lowest free energy is selected as the source of the system conformation to initiate the new simulation window. (In Figure 3.1c, for each of the eight new windows a single window is selected as the source of the initial conformation.).

5. If no window is created in steps 3 and 4, W_{max} is increased by a small increment (until $W_{max} = E_2$) and steps 2-4 are repeated.

6. This process cycles until no more windows can be created within the current free energy barrier limit $W_{max} = E_2$ (or alternatively when a pathway from the initial state of the system to a predefined target state is found).

The self-learning adaptive US calculation can be initiated from a single state of the system as described above. If the final targeted state of the system is already known, the string method can be used to predefine a free energy pathway connecting the initial and final states. In this case, a third parameter Δ_1 is employed to restrict the creation of new windows. A window whose distance from its center to the pathway exceeds Δ_1 will not be added.

3.2 Efficiency of the Self-Learning Adaptive Umbrella Sampling.

For the calculation of PMF in multiple dimensions, one key advantage of methods like ABF[1] and metadynamics [9] is the ability to concentrate the sampling effort to regions of the conformational space that correspond to highest probability density. The convergence of these methods is however dependent on stochastic diffusion along the reaction coordinates for accumulating the required sampling data. On the other hand, conventional implementation of stratified US would typically waste time sampling regions of low probability density, but it is more systematic in its strategy to accumulate data by using the concept of windows. The algorithm we present here combines the advantages of both approaches; i.e., it concentrates the sampling effort to the region of high interest and accumulates data in a systematic way.

This can be illustrated by the argument brought by van Duijneveldt and Frenkel [14] who showed that sampling of narrow windows (steep biasing harmonic window potential) converges more rapidly than that of broad windows (soft biasing potential). This argument also applies to semi brute-force methods like ABF and metadynamics. To flatten a PMF along one dimension using metadynamics or ABF, one needs to sample the length L of this degree of freedom. Diffusion back and forth requires a time $t = L^2/2D$, where D is the diffusion coefficient. Using a stratification procedure, the length L is divided into N windows of width L/N, and the time for diffusion within the window is then:

[1] ABF - Adaptive Biasing Force

$$t_{window} = L^2/2DN^2$$

which goes down like $1/N^2$, much faster than the number of windows. The total theoretical simulation time using a stratified umbrella sampling approach is:

$$t_{US} = Nt_{window} = L^2/2DN = t/N$$

The efficiency gain is thus on the order of the number of windows used. For this reason, ABF simulations are also often subdivided into a number of narrower windows [**?**]. The relaxation time of a diffusing degree of freedom restrained by a harmonic potential goes like k_BT/Dk, where k is the force constant of the biasing harmonic potential. By identification with the diffusion time above, the width of the windows would be $l^2 = \left(L/N\right)^2 = 2\left(k_BT/k\right)$. The larger k is, the shorter is the relaxation time and shorter should be the window width. This is however true only up to a point, depending on slow motions orthogonal to the chosen set of order parameters.

3.3 Test Systems

3.3.1 Model System of Fermat Spiral

We focused here on the ability of the described window creation procedure to follow complex pathways. We performed Monte Carlo simulations on an analytical energy function defined as a Fermat spiral:

$$r = \pm\theta^{1/2}, \theta \in [0, 9/4 \cdot \pi]$$

which was chosen for its non-trivial shape (see Figure 3.2). Sampling was performed in Cartesian coordinate with an initial position set to $[-2, -0.75]$ and the window separation distance to 0.15. Simulation windows were characterized by a biasing harmonic potential

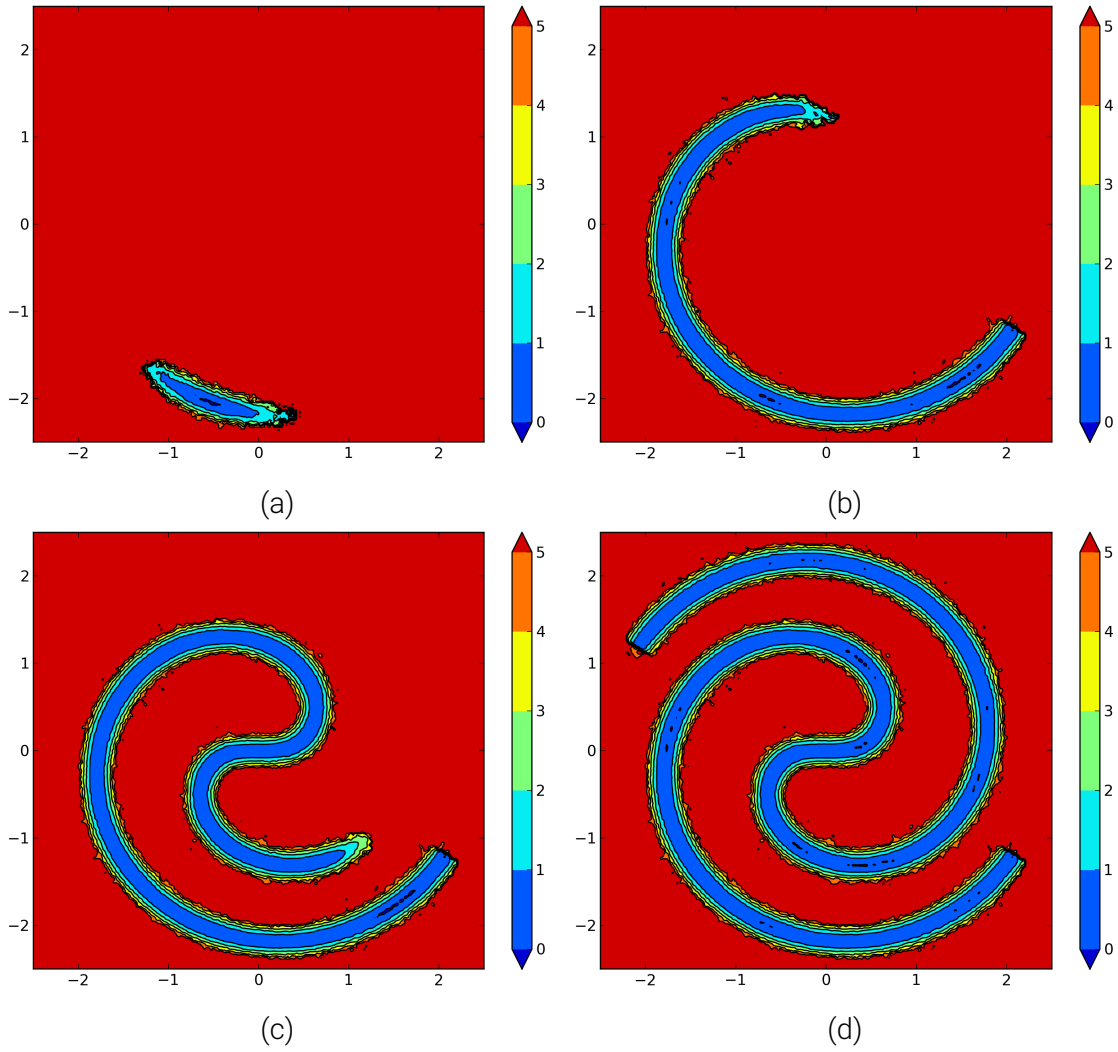


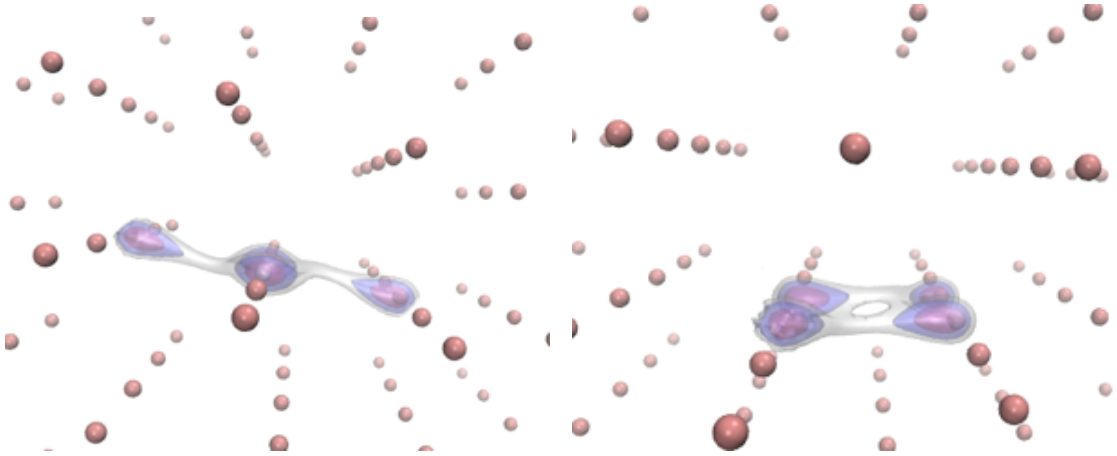
Figure 3.2: Illustration of the self-learning umbrella sampling approach using Monte Carlo simulations and an energy function defined as a Fermat Spiral. The different panels show the energy landscape at different stages of the calculation: (a) Initial energy landscape generated with 9 windows, (b-c) Intermediate stages with 212 and 352 windows respectively (d) The final landscape obtained from 562 windows. The whole space contains 1,156 grid points.

with $k = 6$ kcal/mol · u along both dimensions. Sampling in each window was obtained from a Boltzmann weighted random walk. The WHAM algorithm was applied to calculate the energy landscape from the sampling data. From the initial 9 windows, the automated procedure created new windows in both directions along the spiral function. A total of 562 windows were used to reconstruct the potential function, while the space in which the Fermat Spiral is inscribed contains 1156 grid points (see Figure 3.2).

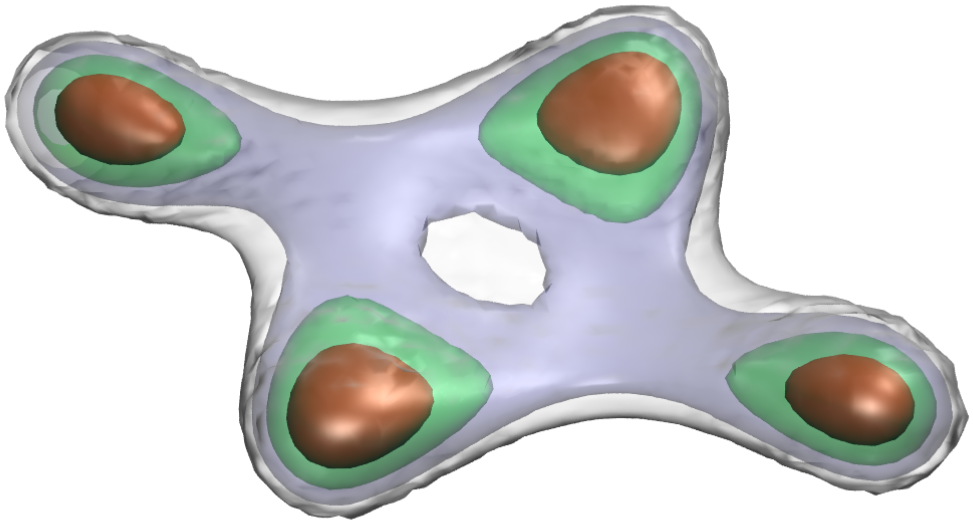
3.3.2 Model System of Lennard-Jones Particles

The self-learning umbrella sampling simulations and WHAM with 3 reaction coordinates was performed to probe a system consisting of Lennard-Jones (LJ) particles. The system consisted of 4 face-centered cubic unit cells of homogeneous LJ particles. The LJ parameters used in our calculations were modified from those of Argon atoms in CHARMM 27 force field, such that $\epsilon = 2 \cdot \epsilon Ar$ and $R_min = R_min, Ar$. All MD simulations were performed using CHARMM [5] molecular simulation program. Canonical ensemble was applied to this model system. The temperature were controlled by Langevin dynamics and kept at 300 K. No cutoff was applied to non-bonded interactions.

Our aim is to describe the displacement of a given particle, number 166, around which three other particles were removed. The Cartesian coordinates of particle 166 were selected as reaction coordinates to calculate the PMF $W[x_1, y_1, z_1]$. A snapshot of the system is illustrated in Figure 3.3. All particles except number 166 were restrained to their "lattice" points by a harmonic potential ($k = 40$ kcal/mol · Å^2). Umbrella windows were generated every 0.5Å in each direction, within a box $[-4, 4] \times [-1, 4] \times [-1, 4]$ Å^3. The force constant of the umbrella potential was 10 kcal/mol · Å^2. Each window was simulated for 100 ps, with a time step of 1 fs. Free energy landscape expansion procedure was initialized with the $E_1 = 2$ kcal/mol and $E_2 = 10$ kcal/mol. The result of this calculation is presented in Figure 3.3 and in Table 3.1.



(a) (b)



(c)

Figure 3.3: Three-dimensional PMF of a system with Lennard-Jones particles: (a) and (b) PMF projection on a system, (c) 3D PMF with contours and 2, 4, 6 and 8 kcal/mol.

	3D PMF
Full configuration space	2057
Essential configuration space, as delimited by the automated procedure	311

Table 3.1: Number of windows required to calculate a PMF describing the process of moving one particle in a LJ-particle system.

Chapter 4

The iPMF Application

In this chapter I will discuss an application that we developed in a purpose of running multiple multidimensional PMF calculations. The starting point for this project was to have an implementation of the Self-Learning Adaptive Umbrella Sampling Method [15], however over the course of my PhD it has evolved far beyond that. iPMF is an application for running potential of mean force calculations and performing analysis of PMFs for various numerical systems. It consists of two layers: the low level C++ implementation for all complex algorithms and data structures, and a Python interface that serves as an user interaction layer. All objects and classes implemented in C++ are exposed to Python, so that every single operation can be performed via a Python script with a native C++ performance.

iPMF includes a support for various molecular dynamics engines, and it can be easily extended to support additional ones (chapter **??**).

Various data manipulation techniques have been implemented; for example, casting data onto spaces with different vector basis (including a reduction and an increase in a number of dimensions), finding the most favorable free energy pathway in a PMF, and more.

4.1 Low level C++ implementation

High performance implementation of any algorithm requires a low level programming language. All of the algorithms and data structures used in the iPMF application were coded in C/C++, and are exposed to Python using boost libraries.

4.1.1 Structure of the multidimensional data

As discussed in chapter 1.3.1, the storage containers for multidimensional data in iPMF are based on sparse matrices. By default those are implemented to support data for up to 6 dimensions. Typical implementation of an array (as a full N-dimensional matrix) is also available, however the number of dimensions is limited to 3. It was mentioned before that the amount of memory required to store a 3D data set can exceed memory of a typical computer, therefore the full arrays should not be used unless there is an explicit need to do that.

4.2 WHAM interface

iPMF uses the Weighted Histogram Analysis Method (chapter 1.2, algorithm 1) to unbias simulation data. The application includes several implementations of WHAM algorithm (as described in chapter 2.1).

4.2.1 iPMF internal WHAM implementation

The default implementation of WHAM follows the description from chapter 2.1.3. It is an internal part of the application, and it can be accessed by a user directly from the iPMF Python shell.

WHAM function is implemented in C++ (definition in chapter **??**) and exposed to Python with the help of boost.python library. This allows the algorithm to run with a full C++ performance directly from within the Python code. WHAM object is converted and exposed

from C++ to Python, it behaves as it would be a part of Python, however all of the algorithm's internal operations are executed without accessing any of the objects registered in Python objects tree (which is the key factor affecting performance of any pure Python implementation).

4.2.2 External MPI WHAM implementation

Another available implementation is the MPI implementation for multi-core systems (chapter 2.1.4). It is accessible as a standalone application that requires a MPI (Message Passing Interface) to be installed on the operating system. To run this application, two files have to be provided: a file containing a list of windows with their complete description (chapter **??**), and a biased density data file (listing 2.9).

```
$ mpirun -np 12 wham windows.list histogram.sdat
```

The application offers an optimal parallel implementation of WHAM algorithm that scales quite well with the number of assigned computational cores. It divides the input data between CPUs and runs WHAM as described in chapter 2.1.4 (listings 2.6, 2.7 and 2.8).

```
number of assigned nodes: 12
total data size: 181126
      data size, node 1:        15094    0-15094
      data size, node 2:        15094    15094-30188
      data size, node 3:        15094    30188-45282
      data size, node 4:        15094    45282-60376
      data size, node 5:        15094    60376-75470
      data size, node 6:        15094    75470-90564
      data size, node 7:        15094    90564-105658
      data size, node 8:        15094    105658-120752
      data size, node 9:        15094    120752-135846
```

```
data size, node 10:        15094    135846–150940
data size, node 11:        15093    150940–166033
data size, node 12:        15093    166033–181126
```

As with most MPI applications, performance may decrease if too many CPUs are used for a problem that does not require it. In those situations, the time for performing calculations for each cycle of WHAM is significantly smaller than the time required for maintaining communication between processing units. Parsing messages between processors becomes a factor that is not only meaningful, but is also a weakest spot in the whole procedure. In certain cases, it might be wise to perform benchmarking to pick the best parameters for WHAM for given system's configuration. This can be done by running the application with a limit of 10 cycles, and picking the most optimal runtime configuration. Doing this might result in huge performance gains, and should not be omitted if algorithm's performance is not satisfactory. Sample result of running such benchmark (presented in figure 4.1) shows that the algorithm performs best when ran on 12 CPUs.

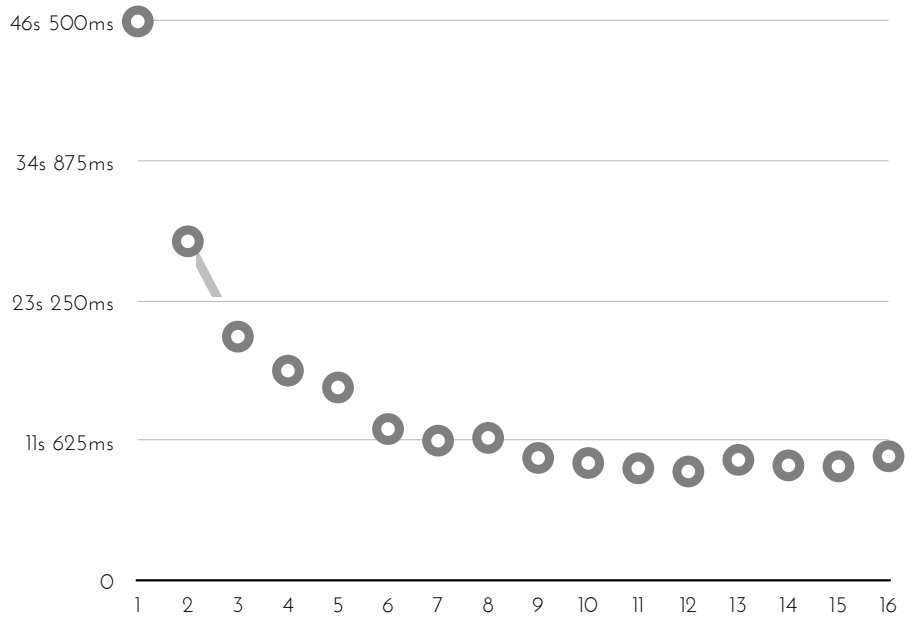


Figure 4.1: Sample result of running a MPI WHAM benchmark. Calculation time of 10 cycles as a function of number of used computational cores.

4.3 Python layer

Python is probably the most broadly utilized scripting language these days. It provides a simple, elegant and intuitive environment that allows performing a vast variety of tasks (ranging from one-line applications to huge, scalable application running on powerful servers). It is a high level interface for interacting with the low level C++ objects, which users can access without having any direct interaction with the low level implementations. Python can be easily extended and embedded in any application, and that is the reason why it became so popular. I like to think about Python as an interface for interacting with all of my C++ code, and I build iPMF application around this concept.

I have designed the iPMF directory structure so that there would be a certain logic behind placement of script files. Name of each directory provides an information about what might be included in this directory. During startup, the main iPMF application loads all of the files included the directories[1]:

- ***config*** - configuration scripts and default settings,
- ***engine*** - functions describing interfaces for various simulation engines,
- ***core*** - core functions responsible for an execution of the self-learning approach,
- ***environment*** - operating system specific functions,
- ***functions*** - commonly used functions,
- ***structures*** - data structures manipulation functions,
- ***xtra*** - additional/external modules.

Each operation performed by a user in iPMF is parsed by a Python interpreter. For the full list of objects and functions that are available in the Python shell, see the iPMF user guide included in the appendix (chapter **??**, page **??**).

[1] Location of iPMF directories is stored in a SQLite database file at: `$HOME/.iPMF/iPMF.db`

4.4 iPMF's internal job scheduler

One of the major problem that I had to address was related to a scheduler that handles the distribution of calculations among available processing units on a computational cluster. In the case of multiple Umbrella Sampling calculations we will be dealing with hundreds of thousands of individual simulations[2]. The obvious problem is a problem of managing all of those simulations on the cluster. A straightforward approach would be to submit everything to a grid engine and let it handle them (figure 4.2), this however might not be the most optimal solution.

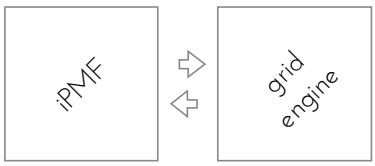


Figure 4.2: Basic model for iPMF interacting with a grid engine.

The idea of implementing an additional layer (that would work as an interface between the iPMF and an existing grid engine) came from answering a fairly simple question: what to do if we would want to adjust priority of simulations that have already been submitted. Having an additional job scheduler between iPMF and a cluster (simulation scheduler, figure 4.3) was the only valid solution, since the grid engines do not usually allow parameter modifications after jobs have been submitted.

The iPMF job scheduler is an application that I designed to run as a daemon on any of the available nodes capable of submitting jobs to a grid. The job of this application is to be an interface between active iPMF sessions and a grid engine. Explicit numerical calculations instead of getting submitted directly to a cluster (figure 4.2) are pre-processed by the scheduler that monitors and manages simulations running on a grid (figure 4.3)).

[2]For a molecular system build with 200 window simulations with 500 steps of 2ps each, 100,000 short ($\pm$1h long) MD simulations will have to be performed.

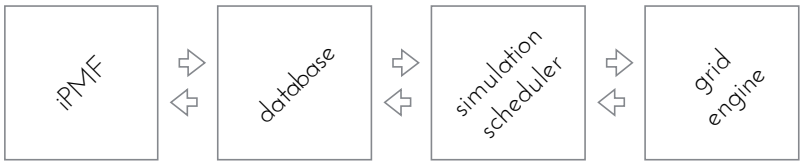


Figure 4.3: Interaction model between iPMF and a grid engine with an additional simulation scheduler in a middle.

To maintain communication between iPMF sessions and a scheduler I have designed a database (figure 4.4) that serves as a necessary interface. An alternative option would be the use of SOCKETS, however that approach can complicate things a bit since the requirement of administrative privileges to set up the interface for network communications.

The scheduler application runs in a background as a daemon, and its job is to: *a*) obtain information about the jobs from a database, *b*) process and manage jobs retrieved from a database, *c*) forward the jobs to a real grid engine according to a predefined algorithm.

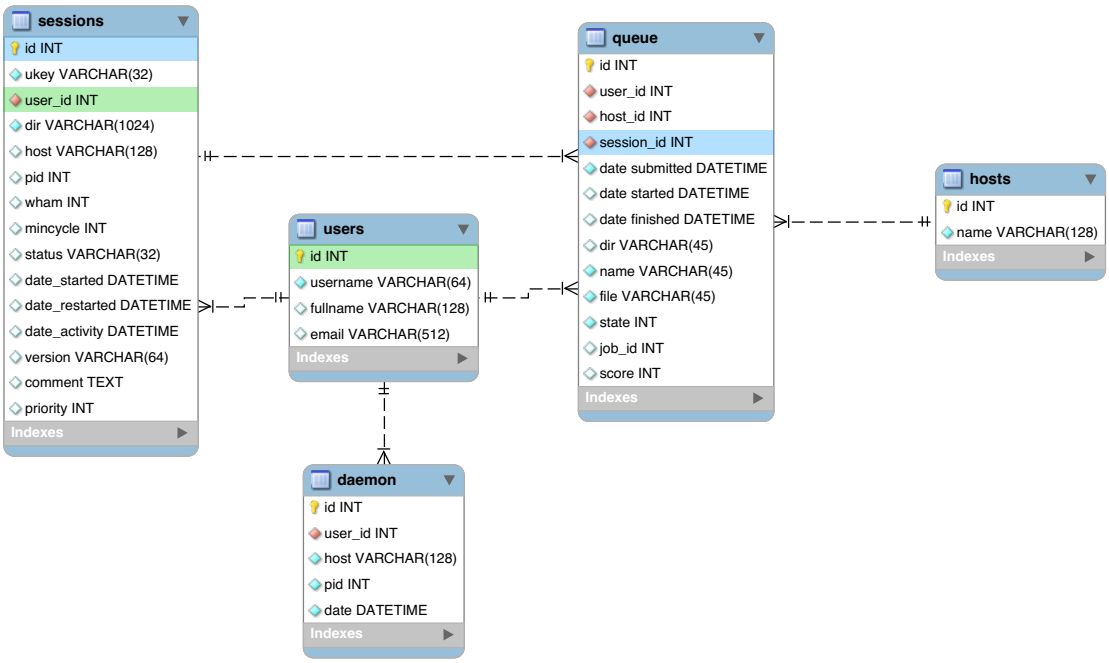


Figure 4.4: iPMF relational database model.

There are two tables in the database that are of importance to the daemon process; one is a **queue** table and the other is a **daemon** table (figure 4.4). The **queue** table contains information about the jobs submitted to a queue: *a*) ***dir*** - working directory for a submitted job, *b*) ***file*** - job file compatible with a grid engine, *c*) ***state*** - the live status of a job ("0" - not processed, "1" - processed by a scheduler, "2" - forwarded to a grid, "-1" - finished), *d*) ***job_id*** - identification number of a job forwarded to a real queue, *e*) ***score*** - parameter that informs a scheduler how to manage execution priority of submitted jobs. The purpose of the **daemon** table is quite different; it was designed to keep track of the running daemons, and to make sure that one user is not running two daemon sessions at the same time (to avoid nonexclusive "write" operations on a database).

iPMF scheduler job management

The most important function of the iPMF scheduler is the management of jobs. It obtains a list of jobs from a database and processes them accordingly. Forwarding of jobs is only possible if there are slots available in a real queue on a grid, and it is limited so that no more than 16 jobs have a "queued" status on a cluster[3]. The initial algorithm for determining the order in which jobs would get forwarded to a grid was a randomized one[4]. Each job was assigned a score based on the priority of the iPMF session that was submitting it:

$$score(simulation.priority) = \textbf{int}\left(random() \cdot 2^{simulation.priority}\right)$$

In this approach the simulation priority is defined as an integer in range from 0 to 10, with a default value of 5. Increasing a value of iPMF session's priority by 1 results in a twofold increase of a probability for its jobs getting forwarded to a grid; jobs with higher scores get forwarded first (with an exception for WHAM processes, which always get

[3] Number of jobs is a parameter that can be adjusted according to needs.

[4] $random() \in [0, 1)$

a maximal score of 1024). The obvious problem with this model is that the jobs which were assigned some very low score values would end up being stuck in a queue for days. To fix that, an additional time dependent component has been added to the equation (the score would get increased the longer simulations stays in the queue):

$$score(\ldots) = \textbf{int}\left(random() \cdot 2^{simulation.priority}\right) + 2^{days(now()-time.submitted)}$$

Even though this is a valid approach, I decided to implement a different one in the final version of the iPMF. Instead of having a random scoring function, a cycle number would be the only parameter responsible for sorting of jobs. Each iPMF session is assigned a dedicated priority queue (implemented as a heap) that keeps a job with the lowest cycle number in the front. The reason behind having separate queues is to maintain the possibility of assigning different priorities to different iPMF sessions. Priority scale remains the same $p \in \{x \in \mathbb{Z} : 0 \leq x \leq 10\}$, and it determines which queue's topmost element would be forwarded to a grid engine first. The process of selecting a queue can be performed as follows: *a*) an execution list is created, *b*) each iPMF session is added to the list $2^{simulation.priority}$ times, *c*) execution list is shuffled *d*) its elements are to be processed sequentially (listing 4.1).

Listing 4.1: Creation of an execution list for the iPMF scheduler.

```
exec = []
for session in sessions:
        for i in range(2**session.priority):
                exec.append ( session )

random.shuffle(exec)

while exec:
        forward(exec.pop())
```

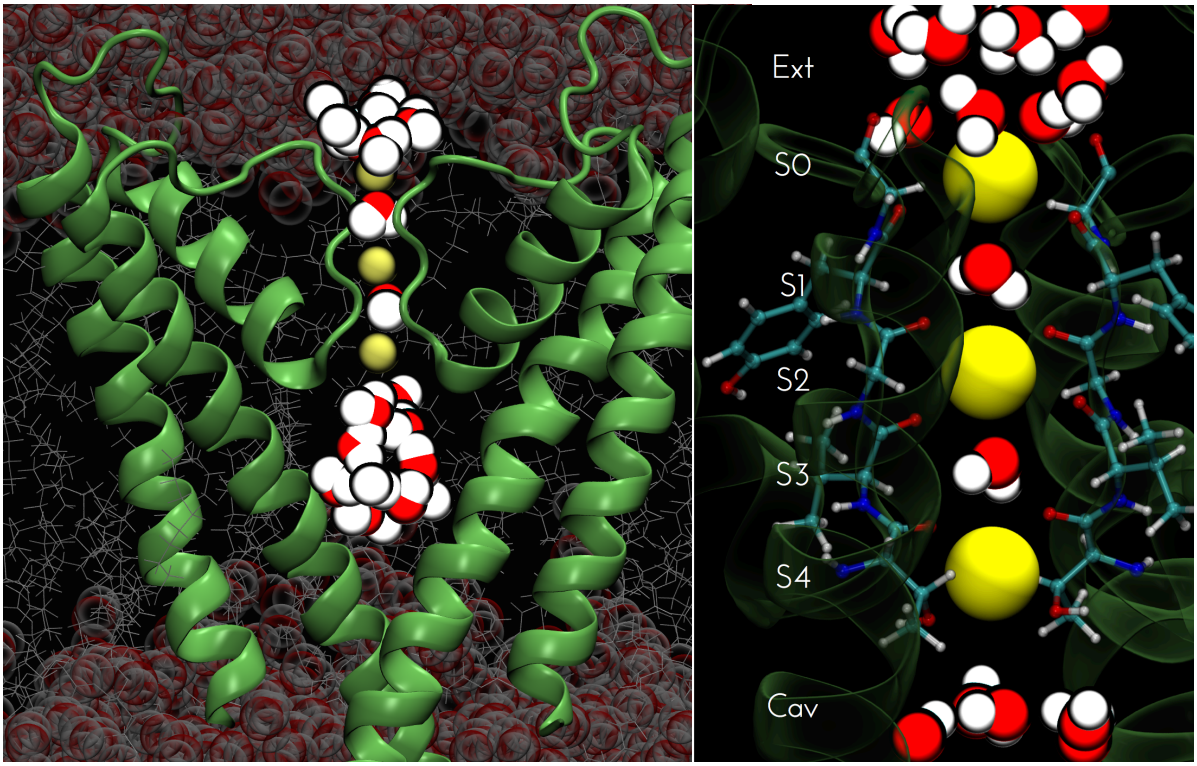
Chapter 5

Molecular Dynamics Simulations

5.1 KcsA potassium channel

Potassium ions diffuse rapidly across cell membranes through proteins called K^+ channels. This movement underlies many fundamental biological processes, including electrical signaling in the nervous system. Potassium channels use diverse mechanisms of gating (the processes by which the pore opens and closes), but they all exhibit very similar ion permeability characteristics (1). Most K^+ channels show a selectivity sequence of $K^+ \approx Rb^+ < Cs^+$, whereas permeability for the smallest alkali metal ions Na^+ and Li^+ is immeasurably low. The amino acid sequence of the KcsA K^+ channel from *Streptomyces lividans* (figure 5.1) is similar to that of other K^+ channels, including vertebrate and invertebrate voltage-dependent K^+ channels. The overall length of the pore is 45 Å, and its diameter varies along its distance. From inside the cell (bottom) the pore begins as a tunnel 18 Å in length (the internal pore) and then opens into a wide cavity (around 10 Å across) near the middle of the membrane [10, 16].

Free energy molecular dynamics calculations on the basis of the X-ray structure of the KcsA K^+ channel have shown that ion conduction involves transitions between two main states, with two and three K^+ ions occupying the selectivity filter. A 'knock-on' mechanism has been proposed, as an model for ion conduction [2].



(a) (b)

Figure 5.1: KcsA K^+ channel in a closed state (Protein Data Bank 1K4C) embedded in a lipid bilayer. Subfigure (b) represents a detailed view of a selectivity filter. Major sites in a filter are labeled: $S0$, $S1$, $S2$, $S3$ and $S4$; Cav represents channel's cavity; Ext is a bulk.

A part of my work was to test the self-learning adaptive method (chapter 3.1) with some explicit molecular dynamics simulations on a KcsA K^+ channel, and to verify the validity of a 'knock-on' mechanism. I have performed various simulations using the iPMF application (chapter 4), and I will go over those in this chapter.

5.2 Closed channel simulations

I started my study of the KcsA K^+ channel by performing some explicit iPMF simulations in the purpose of validating the self-learning adaptive method (chapter 3.1). I used a 1K4C [16] based system with a protein embedded in a DPPC lipid bilayer (156 lipids) with 10780 water molecules (59134 atoms in total). An initial configuration of this system was defined with three potassium ions located in S1, S3 and in cavity (figure 5.1b). I ran some 2D and 3D iPMF calculations; in the 2-dimensional case reaction coordinates were defined as follows:

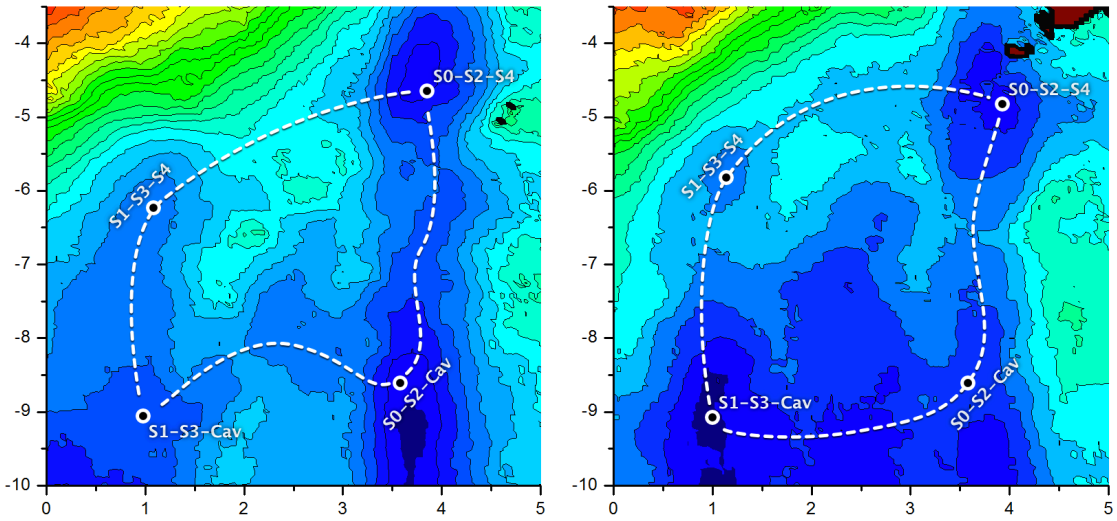
- $Z_{12} = \text{CoM}\,(K_z^1, K_z^2)$
  z coordinate of the center of mass of the top and the middle ion
- $Z_3 = K_z^3$
  z coordinate of the bottom ion

and in the 3-dimensional simulations reaction:

- $Z_1 = K_z^1$
  z coordinate of the top ion
- $Z_2 = K_z^2$
  z coordinate of the middle ion
- $Z_3 = K_z^3$
  z coordinate of the bottom ion

iPMF simulations were performed using the self-learning approach (chapter 3.1) with $E_1 = 2\ kcal/mol$, $E_1 = 8\ kcal/mol$, $\Delta E = 0.5\ kcal/mol$. Expansion run was initialized after every window simulation has generated at least 10 ps of data, or every other 10 ps if no windows were created after the expansion procedure. A sample result of an expansion procedure is presented in the appendix; see figure A.1 (page 72), and figure A.2 (page 73). Molecular dynamics simulations were performed in CHARMM (Chemistry at HARvard Molecular Mechanics) [5] using the CHARMM 27 force field.

The final structure of each potential of mean force has been obtained from the last 50 ps of simulation data (unless stated differently). In all cases the WHAM algorithm (algorithm 1, page 7) has been used for the PMF calculations.



(a) (b)

Figure 5.2: PMFs calculated for the KcsA K^+ channel (1K4C) system: (a) charmm 27 force filed with CMAP corrections enabled, (b) charmm 27 force field with CMAP corrections disabled. Simulations started form a S1-S3-Cavity configuration, with reaction coordinates defined as: $x : Z_{12} = \left(K_z^1 + K_z^2\right)/2$; $y : Z_3 = K_z^3$. Range for values of reaction coordinates was selected so that the energy landscape would include transition pathways from S1-S3-Cavity to S0-S2-S4 configuration. Contours plotted every 1 kcal/mol.

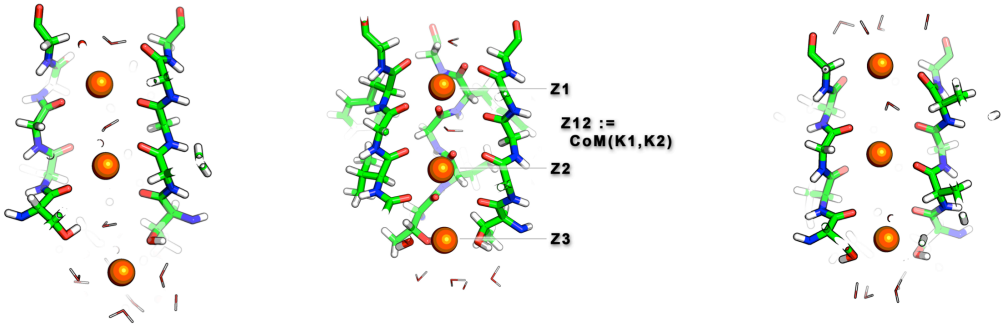
The very first simulation I performed was a copy of a simulation performed by Bernèche and Roux [2]. I rebuild a system using a 1K4C structure (instead of an earlier 1BL8 structure used by Bernèche and Roux [2]), and ran with it an iPMF simulation with a time limit of 100 ps per window. Initial configuration of this system was defined with three potassium ions located in: S1, S3 and in the cavity (figure 5.1b and figure 5.3a). The features of an obtained PMF (figure 5.2a) did not match the ones from an original plot

(figure 2, Bernèche and Roux [2]; figure A.3, page 74), and the only common property was the energy difference between the initial (S1-S3-Cavity) and the final configuration (S0-S2-S4). The system did not behave as we thought it would, and the obtained energy landscape showed ions following a pathway alternative to the one that would correspond to a 'knock-on' mechanism. Further analysis of structures collected at different values of reaction coordinates revealed unphysiological conformational changes in a filter region. Initially we though that this problem might come from an introduction of the CMAP corrections [6] which were not included in CHARMM 22 force filed that was used by Bernèche and Roux [2]. I restarted this simulation using a CHARMM 27 force filed without CMAP corrections[1]. Unfortunately, the result of this simulation did not bring a solution to the problem, see Figure 5.2b. The only apparent difference was a well corresponding to S1-S3-S4 configuration which was not visible in the previous simulation, however the pathway alternative to the one that would be in agreement with a 'knock on' mechanism still seems to be more favorable (due to shallower slopes and smaller energy barriers).

After running numerous simulations, we got to a point when the system was stable and the target PMF was resembling the one obtained by Bernèche and Roux [2], see figure 5.4. The energy barriers were slightly higher than what was expected, however the ion translocation pathway (figure 5.4) was definitely matching the 'knock on' mechanism. This running time was extended to 1 ns per window, and the system remain stable throughout the whole simulation. Figure 5.5d contains a PMF calculated from the last 100 ps of this 1 ns simulation. To have an in-depth view of this simulation I performed an additional 3D iPMF simulation in which each ion was considered separately. This was a 500 ps simulation, and it reveled a structure identical to the one from a 2D iPMF run (compare figures 5.6b and figure 5.6c). Unfortunately, further analysis of this system reveled something unexpected. In an intermediate configuration in a pathway with ions in S1-S3-S4 (see figure 5.4) we found that the water molecule that was originally occupying site S4 was missing (compare figure 5.3a and figure 5.3b). To have another view of the behavior of this system, we ran a long molecular dynamics simulation with the very same starting

[1] CMAP keyword removed from a PSF file.

conformation (figure 5.3a) as described in Boiteux and Bernèche [4]. During the initial 10ns the bottom-most potassium ion (located in the cavity) was attempting to get access to the S4 site. Those attempts were unsuccessful, until a water molecule that was occupying the S4 site moved away to the cavity. Only then there was enough space for the ion to get to the S4 site (figure 5.3b). The "jump" of the other two ions of interest from S1-S3 (figure 5.3b) to S0-S2 (figure 5.3c) occurred almost instantly after the bottom ion was able to move to the S4 site.



(a) (b) (c)

Figure 5.3: Selectivity filter of a KcsA K^+ channel (PDB id: 1k4c), snapshots taken along a translocation pathway of interest: (a) S1-S3-Cavity, water molecules present in S0, S2 and S4; (b) S1-S3-S4, water molecules in S0 and in S2; (c) S0-S2-S4, water molecule in S1, water molecule missing from S3.

The comparison of the initial structures that generated PMFs mentioned in this chapter (see figure 5.2 and figure 5.4), revealed that the only difference was the initial position of the bottom potassium ion. In the first simulation (Figure 5.4) the bottom ion was located slightly below the S4 site, whereas in the other one (figure 5.4) it was located 2 Å below (deeper in the cavity). This small difference allowed a water molecule to leave the S4 site in order to make enough space for the potassium ion (Figure 5.3). At that time we believed that the reason behind this problem was related to the flexibility of the selectivity filter that could originate from an introduction of the CMAP parameters (the reference simulation from Bernèche and Roux [2] was performed with charmm 22 forcefield which

did not include CMAP). However the experiments that we performed next gave a different explanation of this problem.

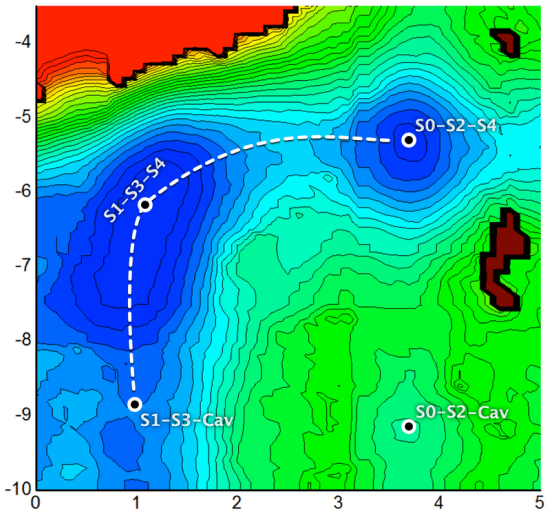


Figure 5.4: PMF calculated using an umbrella sampling method with 154 simulation windows that describes the process of ion translocation in the KcsA K^+ channel (1K4C). Each window was simulated for 100 ps; last 50 ps of sampling was used to calculate this PMF. The difference between this map and the map from a Figure 5.2 is the starting position of the bottom ion. In this simulation, the bottom ion is initially located slightly deeper in the cavity. This allows an ion to enter S4 site after a water molecule that was previously occupying it goes down to the cavity. This map was plotted with a contour every 1 kcal/mol.

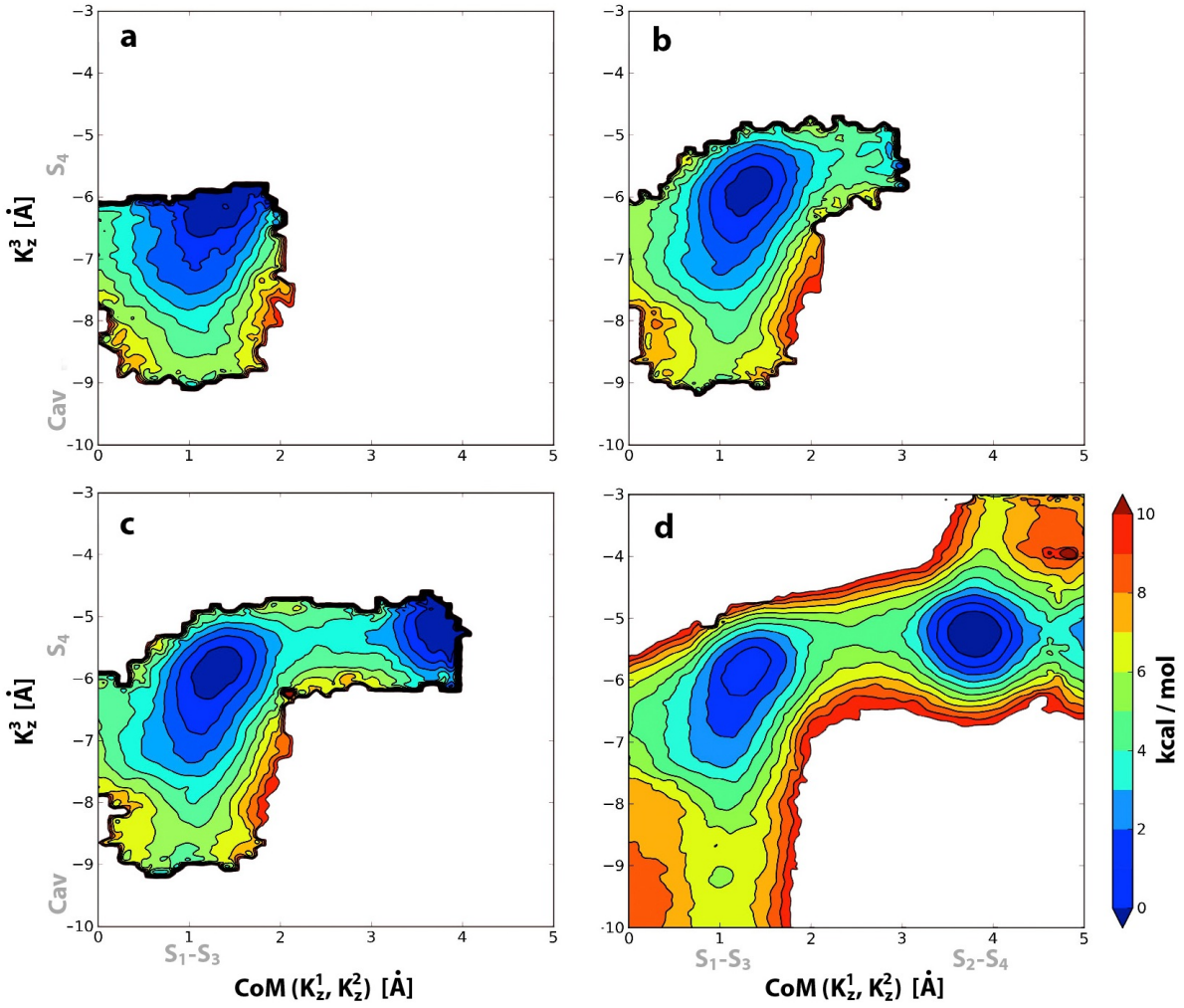


Figure 5.5: Ion translocation in the KcsA K^+ channel described by self-learning umbrella sampling. The 2D PMF is shown at different stages of the umbrella sampling calculations: starting with nine windows (a), moving to 25 (b) and 28 windows (c). The final PMF shown in (d) was calculated from 63 windows. The reaction coordinates are the center-of-mass of ions K^1 and K^2 along the Z axis, CoM (K_z^1, K_z^2), and the position of ion K^3 along the same axis, K_z^3. For the additional information about the expansion procedure please see the appendix: figure A.1 (page 72), and A.2 (page 73).

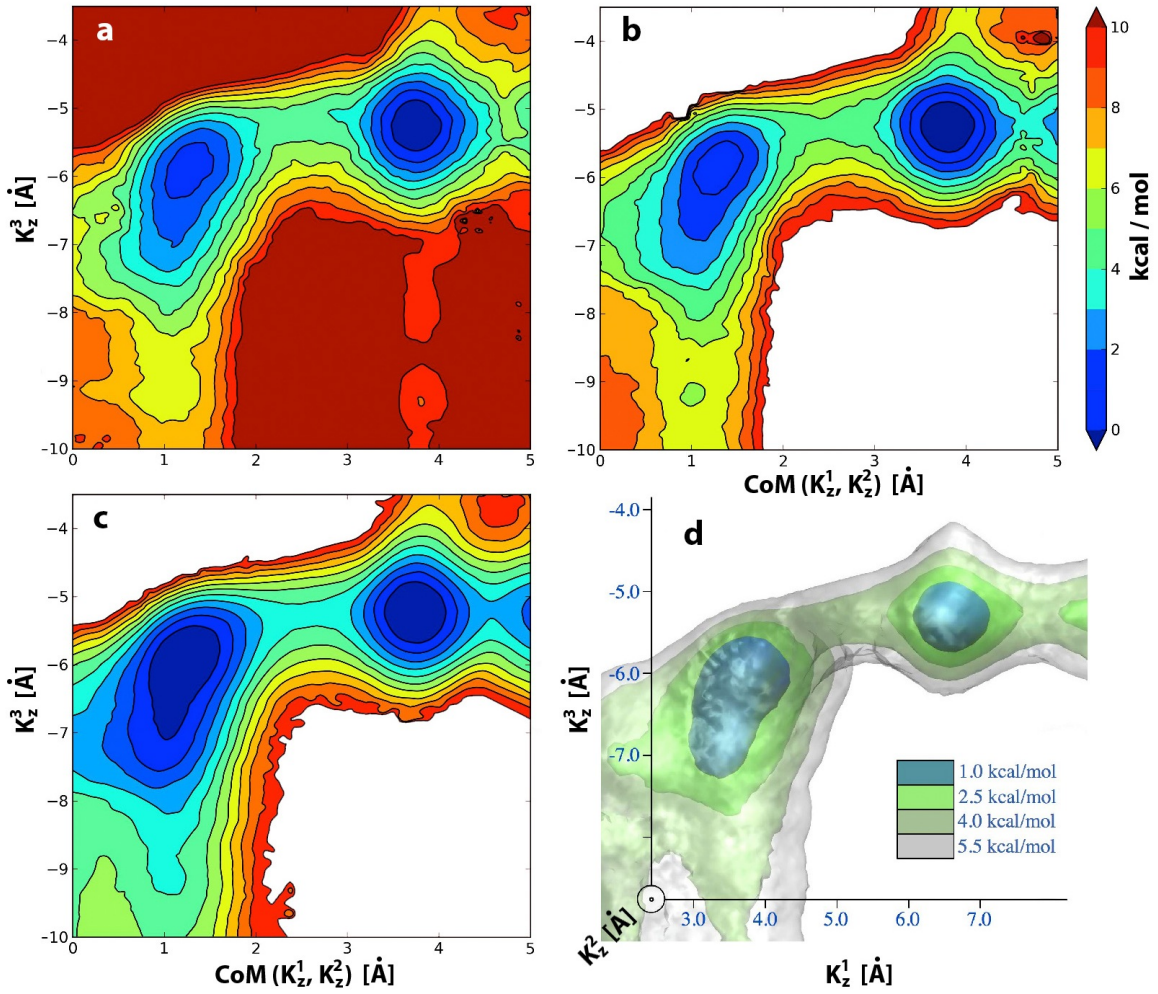


Figure 5.6: Comparison of the ion translocation PMF obtained through different umbrella sampling approaches: (a) 2D umbrella sampling calculation with 154 windows covering the whole conformational space (regions above 10 kcal/mol are not detailed). (b) Result of the 2D self-learning umbrella sampling calculation using a total of 63 windows. (c,d) 2D projection (c) of a 3D PMF (d) calculated with 385 windows generated by the self-learning approach. The reaction coordinates in panels (a), (b) and (c) are as described in a Figure 5.5. In the 3D PMF presented in (d), each ion is considered separately $W\,[K_z^1, K_z^2, K_z^3]$, with K_z^2 sticking out of the plane.

5.3 Open channel simulations

Next set of simulations was based on a KcsA K^+ channel structure with the intracellular gate in the open state (Figure 5.7), PDB entry 3FB7 [8].

Using this system I ran a few iPMF simulations (using the CHARMM 36 force field) with a S0-S2-S4 configuration as a starting point:

1. 2D PMF representing a K^+ ion entering a filter from the cavity (Figure 5.1b), with reaction coordinates defined as follows:

   - $Z_{12} = \text{CoM}\,(K_z^1, K_z^2)$
     z coordinate of the center of mass of the top and the middle ion
   - $Z_3 = K_z^3$
     z coordinate of the bottom ion

2. 2D PMF representing a K^+ ion leaving the filter to the extracellular bulk (Figure 5.1b), with reaction coordinates defined as follows:

   - $Z_1 = K_z^1$
     z coordinate of the top potassium ion
   - $Z_{23} = \text{CoM}\,(K_z^2, K_z^3)$
     z coordinate of the center of mass of the bottom and the middle ion

All of those simulations, were ran with iPMF using the self-learning method for creating new windows. Each window was simulated for 1 ns, however in many cases the system did not remain stable for the last 300 ps. This is an unfortunate problem with that open structure of the KcsA K^+ channel that we are still trying to solve.

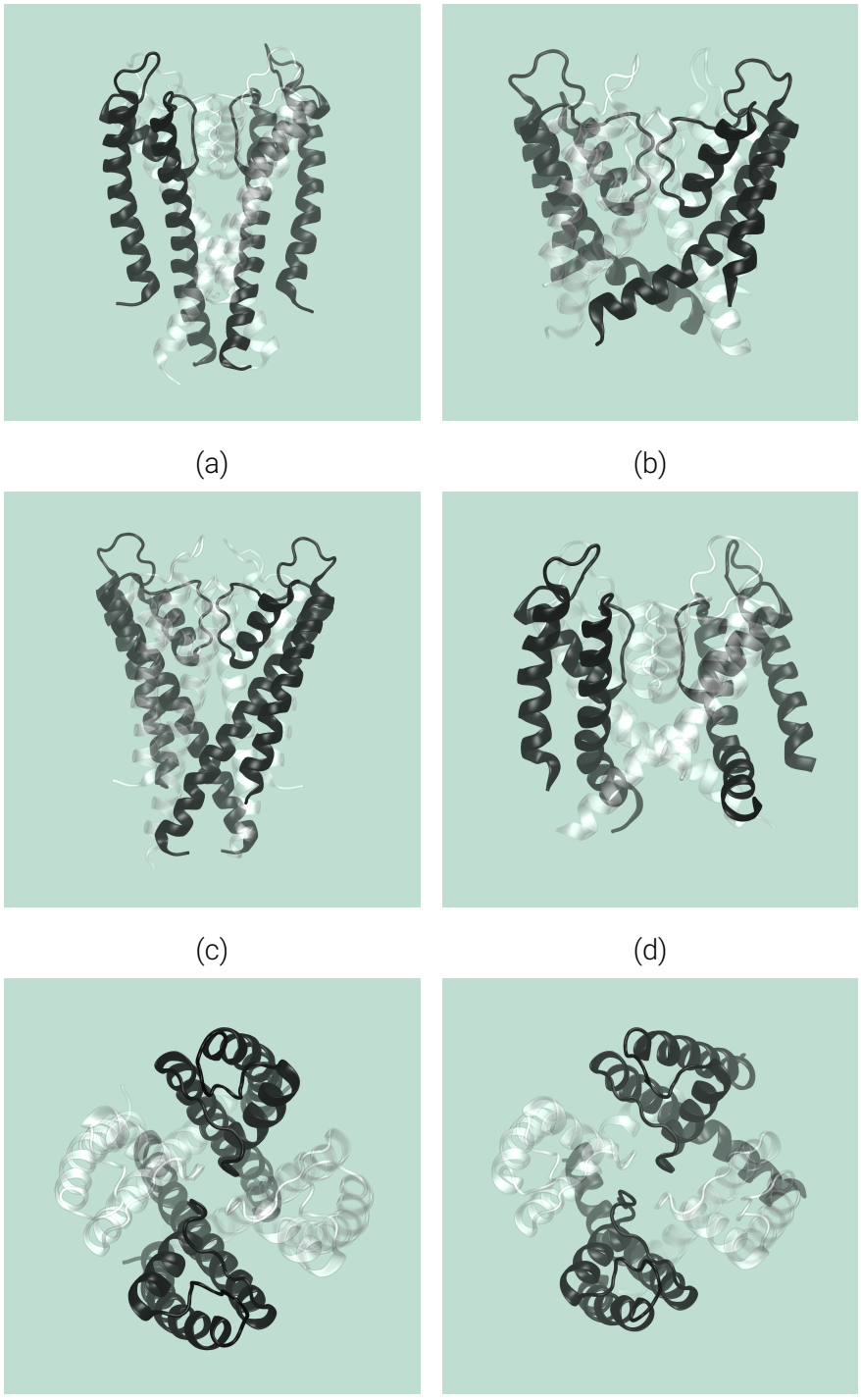
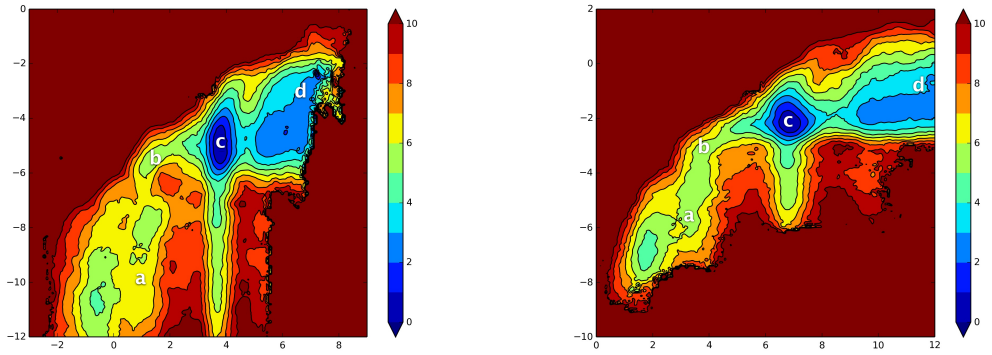


Figure 5.7: KcsA K^+ channel with the intracellular gate in the closed (a,c,e) and in the open (b,d,e) states.



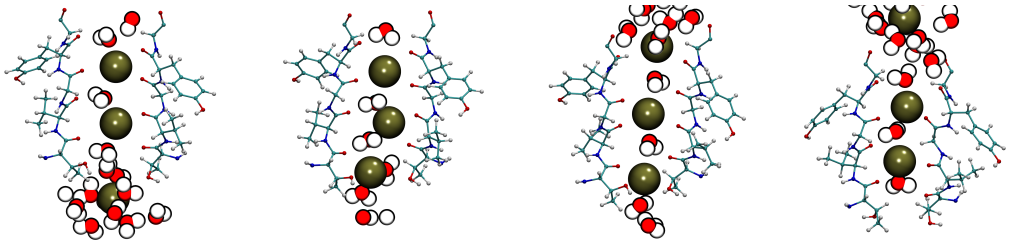
(a) (b)

Figure 5.8: Potential of mean force for a KcsA K^+ channel in an open conformation: (a) a 2D projection of a 3D PMF that represent ion transfer through the channel on the plane $\{x : Z_{12} = \text{CoM}\left(K_z^1, K_z^2\right), y : Z_3 = K_z^3\}$; (b) a 2D projection on the plane $\{x : Z_1 = K_z^3, y : Z_{23} = \text{CoM}\left(K_z^2, K_z^3\right), \}$.

Results of those PMF simulations were merged together into a 3-dimensional energy landscape, and then projected onto appropriate 2D landscapes (Figure 5.8). The main pathway connecting the end-states S1-S3-Cavity and Bulk-S1-S3 includes all of the properties that would support the 'knock on' mechanism, as described by Bernèche and Roux [2]:

1. The **initial state** in the ions transfer process (with ions in S1-S3-Cavity, Figure 5.9a) corresponds to position $(1.0, -9.0)$ on a Figure 5.8a;

2. The bottommost potassium ion enters the filter to site S4 $(1.0, -5.5)$ - see Figure 5.8a, pushing the water molecule that was occupying that site a little to the side (S1-S3-S4 state, Figure 5.9b). Between the S1-S3-Cav and S1-S3-S4 states there is a free energy barrier of around $1\ kcal/mol$;



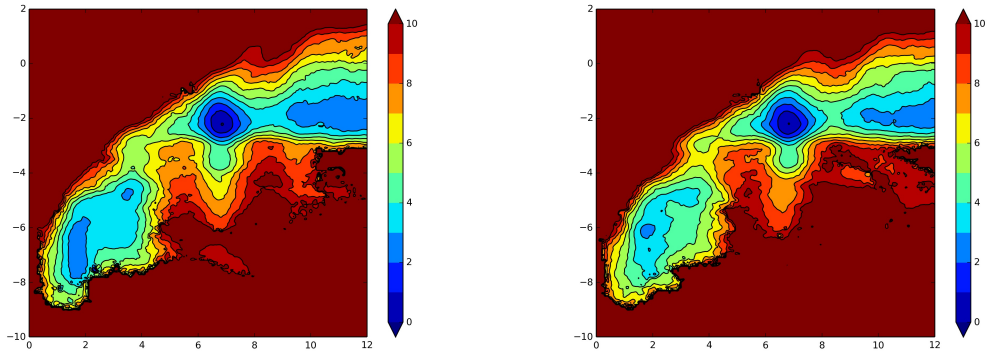
(a) S1-S3-Cav (b) S1-S4-S4 (c) S0-S2-S4 (d) S1-S3

Figure 5.9: Selectivity filter of a KcsA, transfer of ions through the selectivity filter.

3. From state S1-S3-S4 the system goes rapidly to the S0-S2-S4 state (Figure 5.9c). The free energy difference between those states is in range of $-5\ kcal/mol$ (the S0-S2-S4 is the more favorable one) with no apparent free energy barriers. This motion is visible on a free energy profile (Figure 5.8a), and the positions of the corresponding sites are $(1.0, -5.5)$ and $(3.5, -5.0)$;

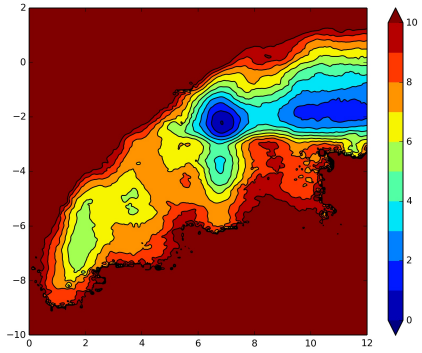
4. In the final step, the system goes from S0-S2-S4 to a state with only two ions occupying the filter in S2-S4. This state changes later to a S1-S3 state, see Figures 5.8b and 5.9d. The motion of the two ions inside the filter is actually visible on the first free energy map (Figure 5.8a). Position: $(-1.0, -11.0)$ describes the system in a S2-S4 state with a third ion deep in the cavity; when the cavity ion pushes up, the two ions move from S2-S4 to a S1-S3 conformation $(1.0, -9.0)$.

The 'knock on' mechanism [2] seems to be present in both the movement of ions from S1-S3-Cavity to S0-S2-S4 (with an intermediate step of S1-S3-S4) and in the movement from S2-S4 to S1-S3 (which seems to be triggered by an additional ion pushing up from the cavity). We believe that the opening of the intracellular gate in the KcsA K^+ channel is the key factor that enables ion transfer through the channel (Figure 5.8).

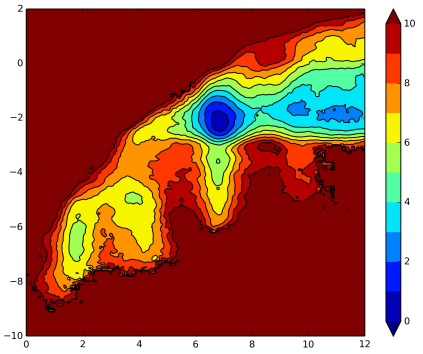


(a) -150 mV

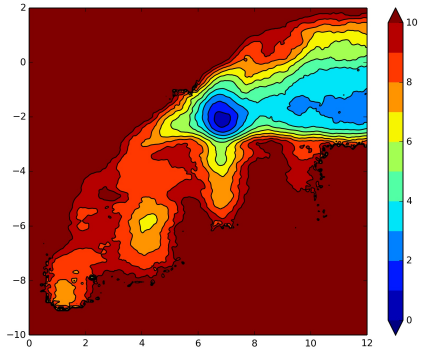
(b) -100 mV



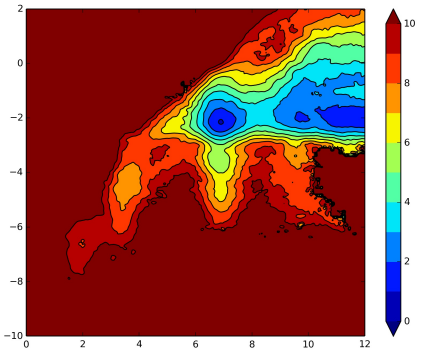
(c) -50 mV



(d) +50 mV



(e) +100 mV



(f) +150 mV

Figure 5.10: Potential of mean force for a KcsA K^+ channel in an open conformation, 2D projections of a 3D PMFs on the plane $\{x : Z_1 = K_z^3, y : Z_{23} = \text{CoM}\left(K_z^2, K_z^3\right), \}$. PMF calculations were performed in the presence of an additional electrostatic potential that was normal to the membrane: (a) -150 mV, (b) -100 mV, (c) -50 mV, (d) +50 mV, (e) +100 mV, (f) +150 mV.

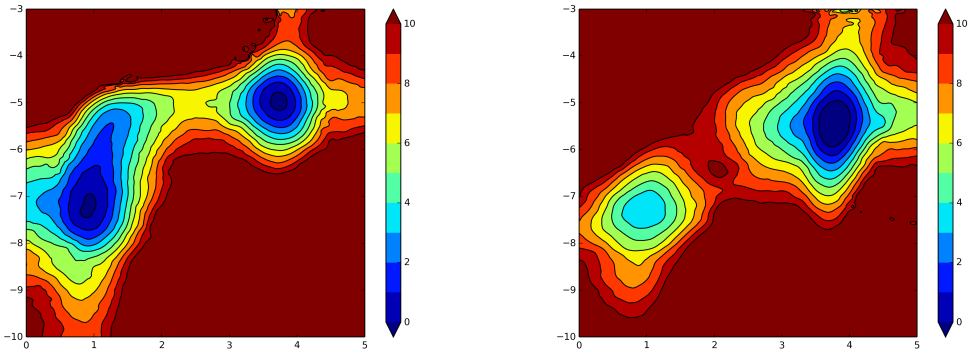
Additional iPMF simulations (with an open structure of a KcsA K^+ channel were performed in a presence of an additional electrostatic potential (normal to the membrane), see Figure 5.10. These results show a trend in which the Ext-S2-S4 conformation gets more and more energetically favorable than the S1-S3-Cav conformation, the bigger is the transmembrane potential applied to the system.

5.4 Summary

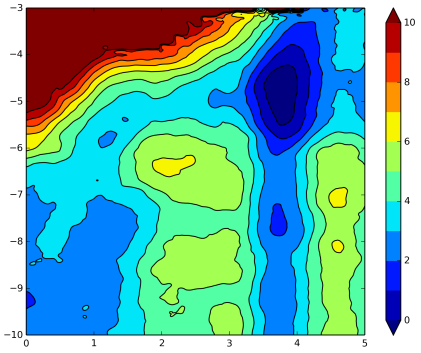
To sum up the KcsA K^+ channel simulations, I will go over some of the most meaningful PMF simulations that relate to the subject of ion translocation.

Result obtained from various simulations of the KcsA channel in a close state, undoubtably show that the channel in this state is incapable of conducting ions. Simulation started with two ions in the filter and additional one in the cavity result in a PMF which is characterized by features that would suggest existence of a 'knock on' mechanism (see Figure 5.11a). However, in-depth analysis of this system shows that this structure does not allow site S4 to be simultaneously occupied by a potassium ion and a water molecule. As a result, system goes into a stable configuration S0-S2-S4 with no water molecule being present in the S3 site. This configuration is separated by a free energy barrier of over 7 $kcal/mol$ from a S1-S3-Cavity configuration.

An attempt was made to force a water molecule not to leave a spot between the middle and the bottom potassium ion. In order to do that, a simulation was started from a S0-S2-S4 configuration and than it was led in the direction of S1-S3-Cavity configuration. The results of this simulation shows a PMF (figure 5.11b) with a high energy barrier, a pathway that does not follow any of the proposed ion translocation strategies, and some unphysiological structural changes in the filter region.



(a) (b)



(c)

Figure 5.11: KcsA K^+ channel iPMF simulations performed in CHARMM 36 force field: (a) 1K4C simulation started at S1-S3-Cavity; (b) 1K4C simulation started at S0-S2-S4; (c) open channel simulation started at S0-S2-S4 (this subfigure represents a time frame of 400 ps to 500 ps of a PMF presented in a figure 5.8a).

Simulation of a channel with an open intracellular gate started in a S0-S2-S4 configuration, results in a PMF (Figure 5.11c) with a translocation pathway that implies a 'knock on' mechanism as described by Bernèche et al. [2]. The S1-S3-Cavity and S0-S2-S4 configurations are separated by a barrier of $\sim 1\ kcal/mol$ which is also in line with previous data.

These simulations suggest that the opening of the intracellular gate is required for the channel to be conductive. We are still working on describing details of this process, and the simulations presented in my dissertation are only a first step in understanding the conducting process of the KcsA K^+ channel.

Chapter 6

Brownian Dynamics

The free energy landscape calculated for the KcsA K^+ channel (structure with the intracellular gate in an open state) provides a description of the process of ion translocation through the selectivity filter (see chapter 5.3; Figure 5.8 and Figure 5.11c). The main pathway connecting the end-states (S1-S3-Cavity and Bulk-S1-S3) includes all of the properties that support the existence of the 'knock on' mechanism, as described by Bernèche and Roux [2]. The problem is that a potential of mean force does not provide us with any direct information about the ion fluxes. Even though the performance of available computational units is increasing every year (as predicted by George E. Moore), we are still not capable of performing molecular dynamics simulations that would be able to visualize ion flux in the KcsA K^+ channel. Time required to run a simulation that could show an ion translocation process is in range of 10-20 ns [12], which can currently be achieved in the matter of a week or two. To have a statistically significant simulation of the ionic flux across the KcsA channel we would require a simulation time thousands times longer than that, therefore it is clear that this is something we cannot afford quite yet.

6.1 Brownian Dynamics Simulations

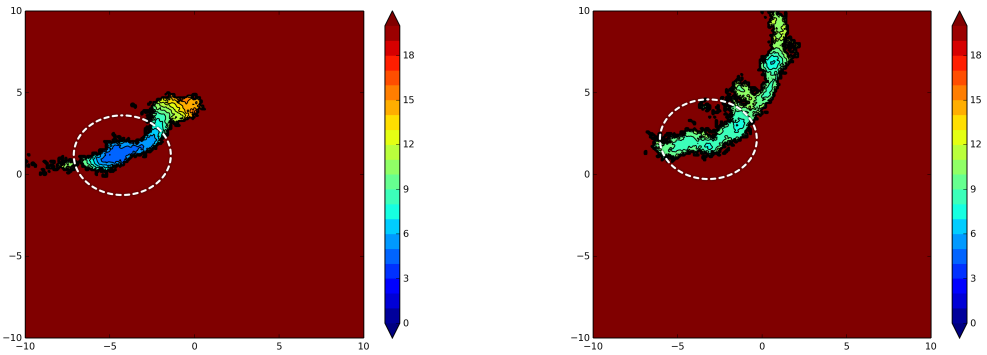
Bernèche and Roux [3] presented a Brownian dynamics (BD) based approach for simulating ion permeation over long periods of time. The result of a BD simulation (ion flux) is the limit (a converged data over a large number of simulation steps) of a function of a free energy landscape and values of reaction coordinates that define it. Any further properties of the molecular system do not have to be explicitly taken into account in a BD approach.

6.1.1 The graph-based BD algorithm

The stochastic Brownian motion for the multi-ion system was first implemented by Bernèche and Roux [3]. It was defined as a continuous-time Markov chain with discrete states corresponding to the ion positions. The state-to-state random walk was constructed by generating exponentially distributed random survival times. I have taken that implementation and decided to reimagine it slightly. In chapter 1.3.2 I introduced an alternative way of representing free energy maps in computer's memory. Applying *graphs* as a main storage data structure in the Brownian motion implementation is extremely beneficial: a) it provides an extremely easy mechanism for adding various additional connections between graphs, b) it removes all of the biological aspects of the system from the random walk procedure - the only part that involves molecular parameters of the system is a definition of edges in a graph. The graph-based approach for performing Brownian dynamics simulations has been implemented in a way that can be used in a virtually any system described by any number of specially separated (though not disjoint) PMFs. I will base my description of this method on a "simple" KcsA K^+ channel system.

During the ion translocation process, the KcsA K^+ channel goes from a state with three ions occupying the selectivity filter to a state with only two ions in the filter. The process of ions permeating through the channel is based on a system jumping back and forth from the three ion state to the two ion state. Complete description of this process requires two graphs, for both a two and a three ion state:

- $G_1(E_1, V_1)$ - graph representing a 3D PMF that describes the three ion state (as shown in chapter 5.3). The 3D PMF has been obtained as a results of merging of two 2D PMFs, see Figures 5.8a and 5.8b;
- $G_2(E_2, V_2)$ - graph representing a 2D PMF that describes a behavior of two ions in a selectivity filter. Taken as a slice from the 3D PMF (at $x = K_z^3 = 10$ Å, Figure 6.1a).



(a) (b)

Figure 6.1: 2D slices of a 3D potential of mean force calculated for a KcsA K^+ channel in an open conformation (see figure 5.8): (a) slice taken at $z = K_z^3 = 10$ Å; (b) slice taken at $z = K_z^1 = -10$ Å. Note, the obvious difference between those figures that derives from different integration constants.

It is important to state that:

$$G_1 \equiv \text{PMF}_{tot}\left(x_1, x_2, x_3\right) \tag{6.1}$$

where:

$$\text{PMF}_{tot}\left(x_1, \ldots, x_{N_D}\right) = \text{PMF}_{eq}\left(x_1, \ldots, x_{N_D}\right) + \sum_{i=1}^{N_D} q\phi_{mp}\left(x_i\right) \tag{6.2}$$

PMF_{eq} denotes the equilibrium PMF, and the $\phi_{mp}\left(x_i\right)$ is the transmembrane potential

function. G_1 and G_2 are not disjoint, since G_2 is defined as a subset of G_1 (figure 6.1). Moreover the relation between those graphs is the following:

$$G_2 \equiv \mathrm{PMF}_{tot}\left(x_1, x_2, 10\text{Å}\right) \approx \mathrm{PMF}_{tot}\left(-10\text{Å}, x_2, x_3\right) \tag{6.3}$$

The graph structure is read from a data file which contains a sparse matrix representation (chapter 1.3.1) of a pre-calculated[1] PMF for the KcsA K^+ channel. Each position in a sparse model corresponds to a node[2] in a graph. Connections between nodes in a graph are managed automatically by the internal functions of a data structure; a self-balancing binary search tree [1, 11] containing a list of all nodes sorted according to their position serves as a interface for finding nodes at the neighboring locations (single query is performed in $O(\log n)$ time, see listing B.1 on page 76). The probability of a system going from a node to one of its neighbors is equal to:

$$\begin{aligned} k_{(x_1,x_2,x_3)\mapsto(x_1\pm\delta x_1,x_2,x_3)} = &\left[\frac{D(x_1) + D(x_1 \pm \delta x_1)}{2\delta x_1^2}\right] \\ &\cdot \exp\left[\frac{\mathrm{PMF}_{tot}\left(x_1 \pm \delta x_1, x_2, x_3\right) - \mathrm{PMF}_{tot}\left(x_1, x_2, x_3\right)}{2k_BT}\right] \end{aligned} \tag{6.4}$$

where D denotes a diffusion coefficient of potassium ions along the selectivity filter.

In the graph model, we can handle data a bit differently. There is no need to give any special treatment to any part of the system, and only one general formula is capable of handling all of the transition rates:

$$\begin{aligned} &k_{(x_1,x_2,x_3)\mapsto(x_1\pm\delta x_1,x_2,x_3)} \equiv \\ &k_{N_i\mapsto\{N_j=\mathrm{neighbour}_m(N_i)\}} = \left\{\sum_{\substack{n=0 \\ n<N_D}} \left[\frac{D\left(x_n(N_i)\right) + D\left(x_n(N_j)\right)}{2\delta x_n^2}\right]\right\} \\ &\qquad\qquad \cdot \exp\left[\frac{\mathrm{PMF}\left(N_j\right) - \mathrm{PMF}\left(N_i\right)}{2k_BT}\right] \end{aligned} \tag{6.5}$$

[1] For the details about PMF calculation check the method described in chapter 3.1 on page 28.

[2] The node is an object containing information about a position and a free energy value, see listing 1.4 on page 11.

Survival time of a state (time a system stays at a given position) in this model is random and distributed exponentially:

$$t_{(i_1,\ldots,i_{N_D})} = {}^{-1}\Big/_{\sum\limits_{\substack{j_1\in[-1,1]\\ \ddots\\ j_{N_D}\in[-1,1]}} \left[k_{(j_1,\ldots,j_{N_D})}\right]} \cdot \log\left(\text{random}()\right) \text{ , random}() \in (0,1) \tag{6.6}$$

Graphs $G_1(E_1, V_1)$ and $G_1(E_2, V_2)$ are connected if:

$$\exists N_{1_i} \in V_1 \wedge \exists N_{2_j} \in V_2 : \left(N_{1_i}, N_{2_j}\right) \in E_s \text{ , } G_1(E_1, V_1) + G_2(E_2, V_2) = G_s(E_s, V_s) \tag{6.7}$$

and:

$$\begin{aligned} G_s(E_s, V_s) = & \\ = \quad & G_1(E_1, V_1) + G_2(E_2, V_2) = \\ = \quad & \left\{ e : e \in E_1 \cup E_2 \cup \left\{ (v_1, v_2) : \begin{array}{c} v_1 \in V_1 \\ v_2 \in V_2 \\ v_{1_x} = v_{2_x} \\ v_{1_y} = v_{2_y} \\ v_{1_y} = +10\text{Å} \end{array} \right\} , v : v \in V_1 \cup V_2 \right\} \end{aligned} \tag{6.8}$$

G_2 is connected with the G_1 via two different sets of connections:

- Ion leaving to the bulk:
  - Find a list L_1 of nodes in G_1 with $x_3 = 10$Å
  - Look for pairs of nodes in L_1 and G_1 that have matching coordinates, and connect them by adding an appropriate edge to $G_1(E_1, V_1) + G_2(E_2, V_2)$

```
std::vector < node<double> * > layer1 =
        G1->query ( std::pair<int,double> ( 2, +10.0 ) );

std::vector < std::pair < int, int > > layer1_connections;
layer1_connections.push_back ( std::pair <int,int> ( 0, 0 ) );
layer1_connections.push_back ( std::pair <int,int> ( 1, 1 ) );

ConnectGraphs <double> (layer1, G2->nodes, layer1_connections, rate_extra);
```

- Ion entering the cavity:
  - Find a list L_2 of nodes in G_1 with $x_1 = -10$Å
  - Look for pairs of nodes in L_2 and G_2 that have matching coordinates, and connect them by adding an appropriate edge to $G_1(E_1, V_1) + G_2(E_2, V_2)$

```
std::vector < node<double> * > layer2 =
        G1->query ( std::pair<int,double> ( 0, -10.0 ) );

std::vector < std::pair < int, int > > layer2_connections;
layer2_connections.push_back ( std::pair <int,int> ( 1, 0 ) );
layer2_connections.push_back ( std::pair <int,int> ( 2, 1 ) );

ConnectGraphs <double> (layer2, G2->nodes, layer2_connections, rate_intra);
```

The detailed implementation B.2 of the *template <typename T> ConnectGraphs<T>* function is included in the appendix on page 77.

After construction of a graph has been completed, an additional function is executed in purpose of checking the integrity of a simulation system. This task can be perforated by querying a graph for disjoint components. This procedure can implemented as a variant of a certain graph traversal methods (*BFS* in my implementation). Usually the largest joint subgraph G_T of $G_1(E_1, V_1) + G_2(E_2, V_2)$ should be used for a simulation.

The main simulation loop in this approach is extremely straightforward, since all of the complex (system specific) interactions define the structure of a graph and not the random walk, the can be implemented in just a few lines:

```
for (int iStep = 0; iStep < NSTEP; iStep++) {
        time += CurrentNode->getSurvivalTime ();
        double r = (double)rand()/(double)RAND_MAX;
        for (int i = 0; i < CurrentNode->probabilities.size(); i++) {
                if (r < CurrentNode->probabilities[i] ) {
                        CurrentNode = CurrentNode->connections[i-1];
                        break;
                }
        }
}
```

Bibliography

[1] R Bayer. Symmetric Binary B-Trees: Data Structure and Algorithms for Random and Sequential Information Processing. 1971.

[2] Simon Bernèche and Benoît Roux. Energetics of ion conduction through the K+ channel. *Nature*, 414(6859):73--77, November 2001.

[3] Simon Bernèche and Benoît Roux. A microscopic view of ion conduction through the K+ channel. *Proceedings of the National Academy of Sciences of the United States of America*, 100(15):8644--8648, July 2003.

[4] Cèline Boiteux and Simon Bernèche. Absence of Ion-Binding Affinity in the Putatively Inactivated Low-[K+] Structure of the KcsA Potassium Channel. *Structure/Folding and Design*, 19(1):70--79, January 2011.

[5] B R Brooks, C L Brooks III, A D Mackerell Jr., L Nilsson, R J Petrella, Benoît Roux, Y Won, G Archontis, C Bartels, S Boresch, A Caflisch, L Caves, Q Cui, A R Dinner, M Feig, S Fischer, J Gao, M Hodoscek, W Im, K Kuczera, T Lazaridis, J Ma, V Ovchinnikov, E Paci, R W Pastor, C B Post, J Z Pu, M Schaefer, B Tidor, R M Venable, H L Woodcock, X Wu, W Yang, D M York, and M Karplus. CHARMM: The biomolecular simulation program. *Journal of Computational Chemistry*, 30(10):1545--1614, July 2009.

[6] M Buck, S Bouguet-Bonnet, and R W Pastor. Importance of the CMAP correction to the CHARMM22 protein force field: dynamics of hen lysozyme. *Biophysical journal*, 2006.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to algorithms. pages 221--252, 2001.

[8] L.G. Cuello, V. Jogini, D.M. Cortes, A.C. Pan, D.H. Gagnon, J.F. Cordero-Morales, S. Chakrapani, B. Roux, and E. Perozo. Open KcsA potassium channel in the presence of Rb+ ion. *Not published yet*.

[9] Eric Darve and Andrew Pohorille. Calculating free energies using average force. *The Journal of Chemical Physics*, 115(20):9169--9183, November 2001.

[10] D A Doyle. The Structure of the Potassium Channel: Molecular Basis of K+ Conduction and Selectivity. *Science*, 280(5360):69--77, April 1998.

[11] E. M. Landis G. Adelson-Velskii. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263--266, 1962.

[12] Meredith LeMasurier, Lise Heginbotham, and Christopher Miller. Kcsa It's a Potassium Channel. *The Journal of General Physiology*, 118(3):303--314, September 2001.

[13] Benoît Roux. The Calculation of the Potential of Mean Force Using Computer-Simulations. *Computer Physics Communications*, 91(1-3):275--282, September 1995.

[14] J. S. van Duijneveldt and D. Frenkel. Computer simulation study of free energy barriers in crystal nucleation. *The Journal of Chemical Physics*, 96(6):4655--4668, 1992.

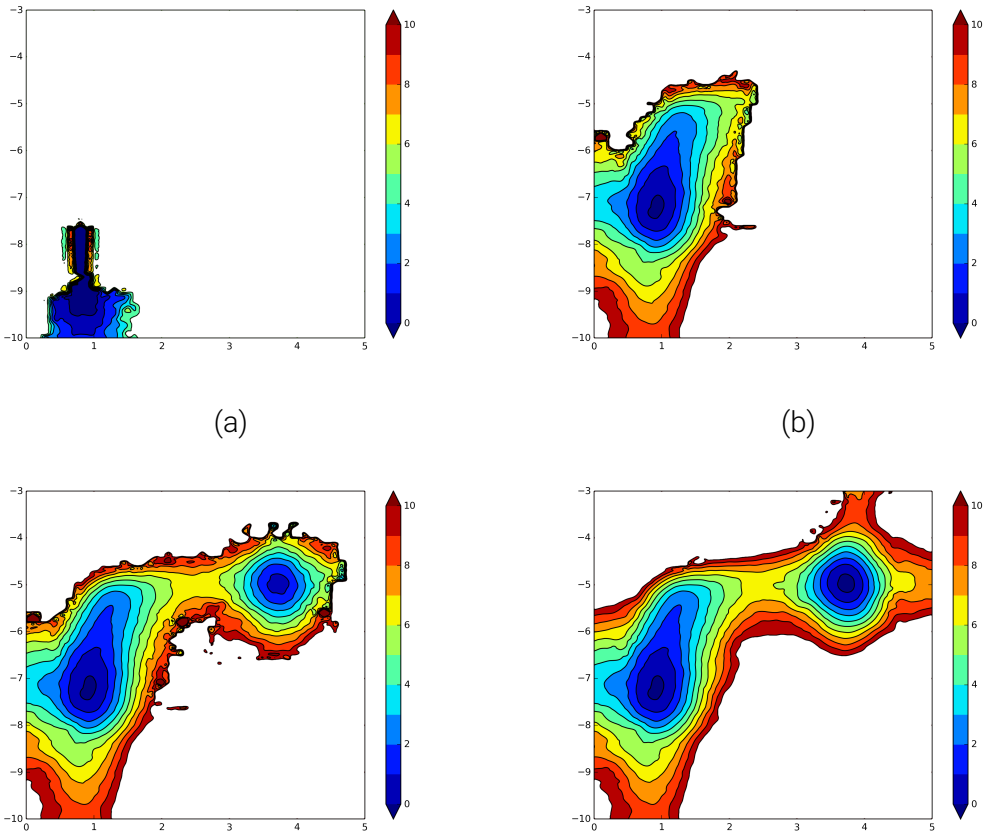
[15] Wojciech Wojtas-Niziurski, Yilin Meng, Benoît Roux, and Simon Bernèche. Self-Learning Adaptive Umbrella Sampling Method for the Determination of Free Energy Landscapes in Multiple Dimensions. *Journal of Chemical Theory and Computation*, 9(4):1885--1895, April 2013.

[16] Y Zhou, J H Morais-Cabral, A Kaufman, and R MacKinnon. Chemistry of ion coordination and hydration revealed by a K+ channel–Fab complex at 2.0 Å resolution. *Nature*, 414(6859):43--48, 2001.

Appendices

Appendix A

Additional Figures



(a) (b)

(c) (d)

Figure A.1: iPMF expansion procedure steps for KcsA (Protein Data Bank 1K4C) calculated in CHARMM36 force filed:
(a) 6 windows; (b) 58 windows; (c) 87 windows; (d) 107 windows.

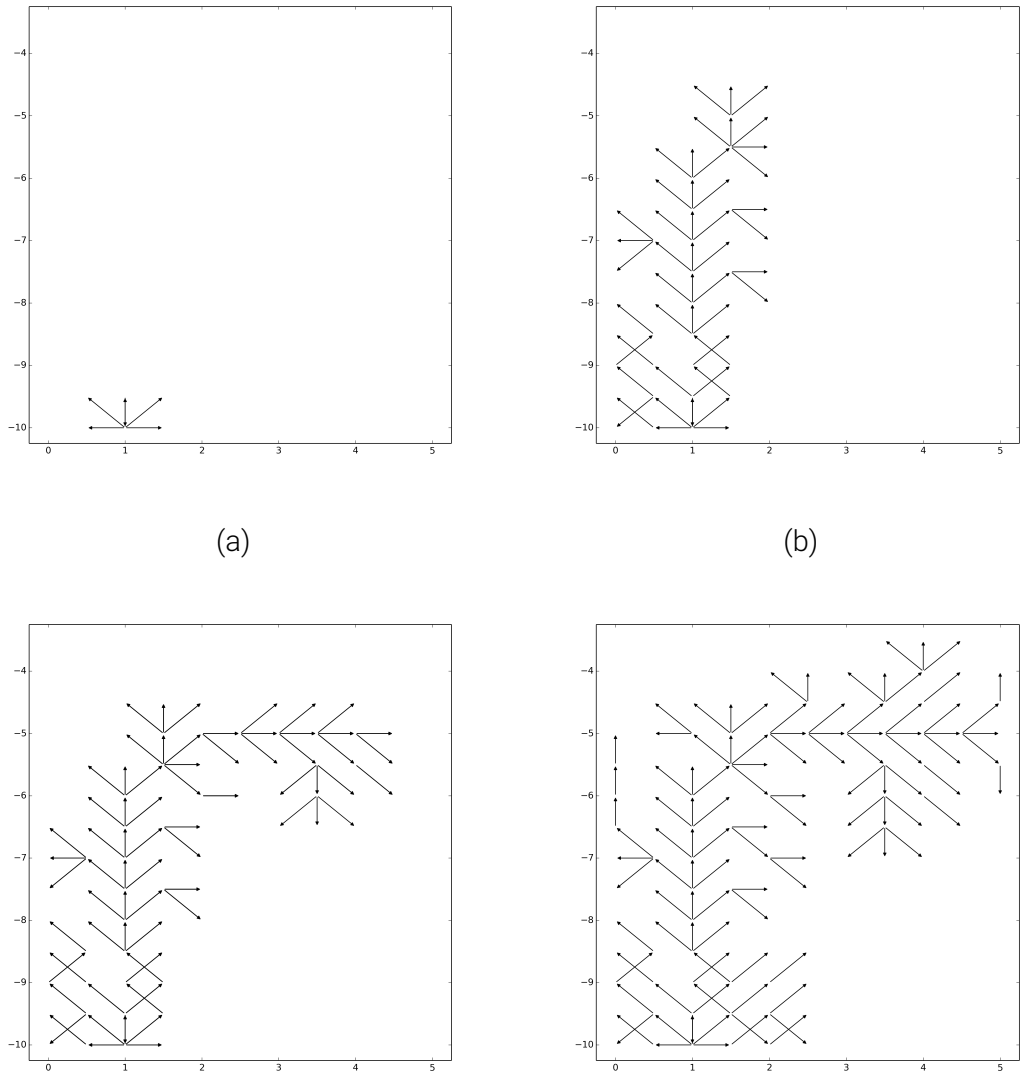


Figure A.2: iPMF expansion procedure steps for KcsA (Protein Data Bank 1K4C), the hierarchy of windows: (a) 6 windows; (b) 58 windows; (c) 87 windows; (d) 107 windows.

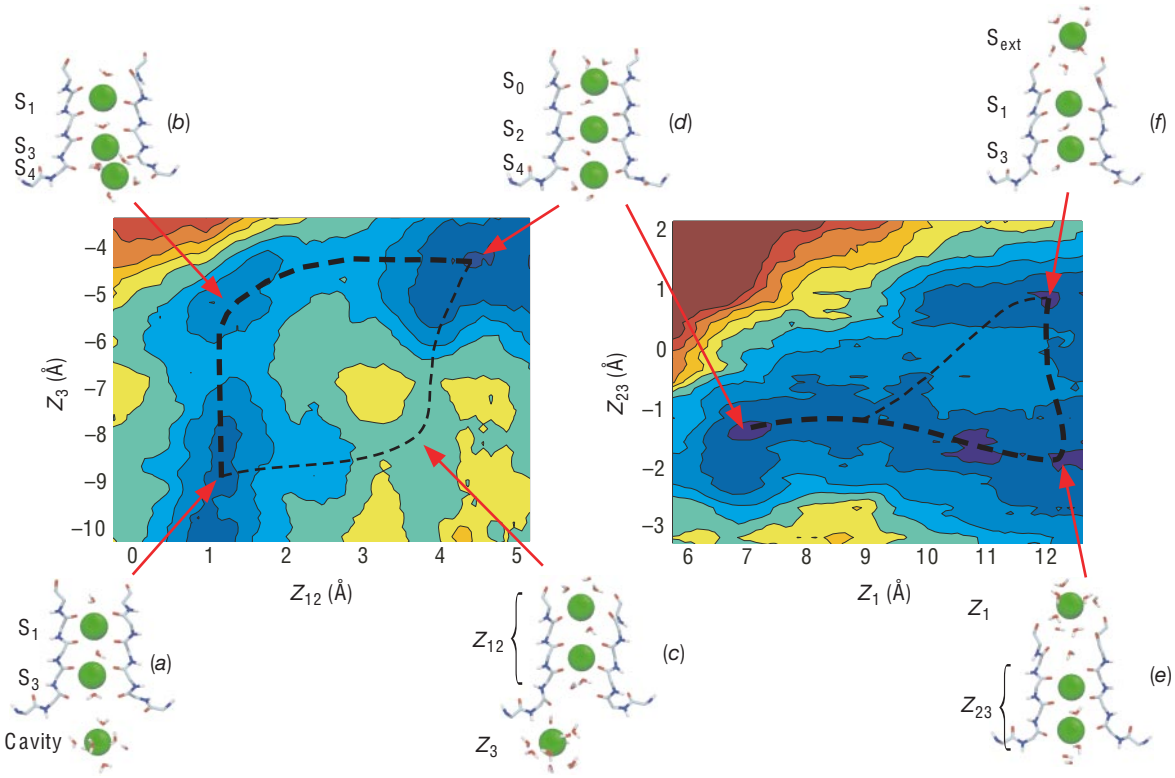


Figure A.3: Bernèche et al. [2]: Topographic free energy maps of ion conduction calculated from umbrella sampling molecular dynamics simulations.

Listing B.1: Connecting a node in a graph with its neighbors.

```
template <typename T>
void graph <T> :: connect ( node<T> * obj ) {

    // number of dimensions
    int N = obj->coordinates.size();

    // get a position of a node
    std::vector < int >             crds = obj->coordinates;

    typename std::map < std::vector<int> , node<T> * > :: iterator query;

    for (int i=0; i<N; i++) {

        // check one at x_i + 1
        crds[i] = obj->coordinates[i] + 1;
        if ((query = this->lookup.find ( crds )) != this->lookup.end()) {
            obj->connections.push_back ( query->second );
            query->second->connections.push_back ( obj );
        }

        // check one at x_i - 1
        crds[i] = obj->coordinates[i] - 1;
        if ((query = this->lookup.find ( crds )) != this->lookup.end()) {
            obj->connections.push_back ( query->second );
            query->second->connections.push_back ( obj );
        }

        // reset the variable of interest
        crds[i] = obj->coordinates[i];
    };
};
```

Listing B.2: Connecting two sets of nodes together: a) finding a list of nodes that share predefined permutation of values of coordinates; b) connecting them with each other; c) each connection is represented by two edges.

```
template <typename T>
void ConnectGraphs ( std::vector < node<T> * > G1, std::vector < node<T> * > G2, std::vector < std::pair<int,int> > connections, double rate ) {
    // building a bst using data from the first list of nodes
    std::map < std::vector <int> , node<T> * > query;
    for (int i = 0; i < G1.size(); i++) {
        // definition of a "reduced" key
        std::vector <int> key;
        for (int j = 0; j < connections.size(); j++)
            key.push_back ( G1[i]->coordinates [connections[j].first] );
        query [key] = G1[i];
    };
    // finding nodes G2 that would match nodes in G1
    typename std::map < std::vector <int> , node<T> * > :: iterator query_iterator;
    for (int i = 0; i < G2.size(); i++) {
        std::vector <int> key;
        for (int j = 0; j < connections.size(); j++)
            key.push_back ( G2[i]->coordinates [connections[j].second] );
        query_iterator = query.find ( key );
        if (query_iterator != query.end()) {
            // connect graph #1 with graph #2
            query_iterator->second->connections.push_back    ( G2[i] );
            query_iterator->second->probabilities.push_back  ( rate );
            // connect graph #2 with graph #1
            G2[i]->connections.push_back    ( query_iterator->second );
            G2[i]->probabilities.push_back  ( rate/45 );
        };
    };
};
```