Internal Report 2012-2013–17 August 2013

Universiteit Leiden
Opleiding Informatica

A kinetic Monte Carlo implementation
of the Cellular Potts Model
with SciQL

Mathé Zeegers

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

1 Abstract

The Cellular Potts Model (CPM) is a widely-used, stochastic method for modeling and simulating collective cell behavior in biology. It has been applied to blood vessel growth, somitogenesis and tumor growth. The main disadvantage of CPM is the lack of a well-defined time scale. In this project a kinetic variant of the CPM will be implemented inspired by kinetic Monte Carlo methods, like the one that has been used for the Gillespie algorithm. In the forthcoming implementation, complex, grid based datastructures and neighborhood queries are needed. For this reason, the model will be implemented with SciQL (http://www.scilens.org/Resources/SciQL) in coorporation with the Database Architectures group at CWI. SciQL is an SQL-based database language meant for scientific applications and has the right syntax for such complex queries on tables and arrays. Experiments with the implementation yield plots where the Hamiltonian is set against the theoretically obtained time scale. The relations between crucial parameters of the CPM, the compression resistance of cells and the temperature of the model, and the simulation in terms of the total energy in the system have been analyzed. The behavior of the implementation corresponds to the theoretical behavior. The temperature determines roundness and the amount of inlets of cells and the compression resistance determines the flexibility of cells.

Contents

2 Introduction

2.1 Cellular Potts Model

To model the collective behavior of cells, Glazier and Graner developed their Cellular Potts Model (CPM) in 1992 [1]. This model takes the adhesion between the cells as a base for the rearrangement of cells, which is determined stochastically.
This model is powerful and relatively easy to understand and is therefore used in a wide range of biological fields, including growth of blood vessels [2], somitogenesis [3] and tumor growth [4]. Some biological mechanisms involved are chemotaxis (the movement of single-cell or multicellular organisms according to chemicals in the environment), haptotaxis (directed motility of cells by an adhesive gradient) and haptokinesis (adhesive protein-mediated motility of cells). Depending on the problem one is facing, the model can be extended in various ways. For example, one may consider to add an extra component that takes the change of size of the surface of the membrane in the CPM into account.

The Cellular Potts Model is a Cellular Automaton (CA) that considers a lattice on which different kinds of cells are living. In the Cellular Potts Model, these cells are represented as clusters of sites with the same identity σ and they try to copy parts of themselves to parts of their neighbors. The process of copying costs or produces a certain amount of energy. This energy describes the work associated with the extension of cells. The change in energy determines the probability that the copy will take place. A copy will also be called an 'update' later on. For describing the most important update rule in the CPM, we need to address the parameters that are used. First of all, consider an environment which consists $n-1$ different cells and one type of medium. The environment is abstracted by laying down a grid on the cells. In this way, we obtain a two-dimensional array in which the elements carry the value of the cell they belong to. Thus, the identity of an element is denoted by $\sigma(\vec{x})$, where $\vec{x} \in \Omega \subset \mathbb{Z}^2$ is an element from the array and $\sigma(\vec{x}) \in \{0, ..., n\}$. Sometimes this value is called the spin. The medium has spin 0. Each spin corresponds to a certain type of the cell it belongs to, which is given by $\tau(y)$, where y is a spin.
Since each pair of cells has a certain adhesion, we introduce the $n \times n$-matrix J in which the adhesion energies are given to model the adhesion of cells to one another. The strength of the area constraint of the cells is given by λ. Other important parameters are α_σ and A_σ. α_σ denotes the current area of a certain cell σ and A_σ the target area that a cell σ tries to obtain ($\sigma \in \{1, ..., n\}$).

Also, consider the Kronecker delta, which is defined by:

$$\delta_{ij} = \begin{cases} 1 & \text{when } i = j, \\ 0 & \text{when } i \neq j. \end{cases}$$

The mentioned change in energy is the total energy after the copy minus the total energy before the copy. The energy is given by the Hamiltonian H, so:

$$\Delta H = H_{after} - H_{before}$$

where H is given by:

$$H = \sum_{\vec{x},\vec{x}'} J_{\tau(\sigma(\vec{x})),\tau(\sigma(\vec{x}'))} \cdot (1 - \delta_{\sigma(\vec{x}),\sigma(\vec{x}')}) + \lambda \sum_{\sigma} (\alpha_\sigma - A_\sigma)^2$$

Let $\vec{x}$ and $\vec{x}'$ be adjacent if $\vec{x}'$ is one of the eight neighbors of $\vec{x}$ on the grid. All adjacent pairs $(\vec{x}, \vec{x}')$ are considered and the corresponding adhesion energies are added. The Kronecker delta cancels out the pairs from the summation where the elements have the same corresponding cell. Since the stretching of cells also contributes to the total energy, the last terms takes care of the sizes of the surfaces of all cells.
Once ΔH is known, the probability of a copy can be calculated. This is done according to the Boltzmann-Gibbs distribution. If ΔH is less than zero, the copy produces energy. This means that such a copy will be executed by all means, so the probability is 1. If ΔH is equal or higher than zero, then the copy does not produce energy, but, in case of ΔH higher than 0, the copy will cost energy. In this case, such a copy will be executed with an exponential probability that depends both on ΔH and on $T > 0$, which is the temperature. This yields the following:

$$\mathbb{P}(\Delta H) = \begin{cases} e^{-\frac{\Delta H}{T}} & , \Delta H \geq 0, \\ 1 & , \Delta H < 0. \end{cases}$$

Now we take a closer look at role of the crucial parameters in this Hamiltonian. The entries of the matrix J can be changed depending on what cells one is working with. This change has a big impact on the Hamiltonian H if the size of the cell is high. If λ increases, the stretching of the cells will be penalized by a higher contribution of this term to the Hamiltonian. This also goes for cells that shrink. As a consequence, a higher λ results in less variation in cell sizes from their respective A-values. The temperature T determines the probability $\mathbb{P}(\Delta H)$ in case $\Delta H > 0$. An higher T increases $-\frac{\Delta H}{T}$ and, therefore, increases the corresponding exponential probability $\mathbb{P}(\Delta H)$ of $\Delta H > 0$. This relates to the fact that cells that are close to their target

area will generally get a round shape when the temperature is low and a more random shape when the temperature increases. In the latter situation, there are more possible copies in one step.

2.2 Motivation for a new algorithm

The easiest way of implementing this model is by using a standard Metropolis algorithm [5]. In this way, at every step we first take a random element from the array. After this, one of his eight neighbors is selected randomly. For the copy corresponding to this pair the Hamiltonian values are calculated, along with the probability. Comparing the value that results from the latter with a randomly generated number from $[0, 1]$, the copy will be executed or not. After this, the process starts again with a new step.

The implementation of the model with the standard Metropolis algorithm as described in the previous section has drawbacks. In this algorithm, many possible updates (or copies) are generated and tested. Unfortunately, many of them are thrown away because the cell types are equal or the probability is not high enough for acceptance. This is waste of computation time. Furthermore, in this implementation there is no good definition of a time scale. This means we are not able to attach a speed to the process and we cannot know at what rate cells make copies. In the end, it should be possible to determine the rate at which pseudopodia are extended and retracted. With this information, the speed of the algorithm can be tuned by setting the parameters correctly using experimental data.

3 Methods

3.1 The new algorithm

The new implementation is a kinetic Monte Carlo (KMC) algorithm and takes away the major disadvantages of the Metropolis implementation. In general, kinetic Monte Carlo methods simulate natural processes. When all parameters are correct and the processes, which occur at certain rates, are independent, a kinetic Monte Carlo algorithm gives a meaningful time scale. There are many variations on the kinetic Monte Carlo method, like the Gillespie algorithm [7], but they mainly differ in their applications. For example, A. Neagu et al. presented the kinetic Monte Carlo method as an

alternative for the Metropolis Monte Carlo for simulations of multicellular systems [6].

In our problem, we first consider all possible updates that can take place in the current state of the model along with the corresponding values of ΔH. These are all stored in a database. Then, if i is a record in this database, the probability that this update will take place is calculated and renamed to $a_i := \mathbb{P}(\Delta H_i)$. Now we obtain a distribution where update i is more likely to be chosen if a_i increases. Furthermore, we define:

$$a_0 := \sum_i a_i$$

The rules for the algorithm are derived below. Firstly, we define the probability density function $\mathbb{P}(\tau, \mu)$ by:

$$\begin{aligned} \mathbb{P}(\tau, \mu)d\tau &= \text{the probability that given time } t\text{, the next} \\ &\quad \text{update will happen in the interval } (t+\tau, t+\tau+d\tau), \\ &\quad \text{and that this update is corresponding to } a_\mu \end{aligned}$$

Now $a_\mu dt$ is the probability that, given time t, the corresponding update will be the update happening in $(t, t+dt)$. Let:

$$\begin{aligned} \mathbb{P}_0(\tau)d\tau &= \text{the probability that given time } t\text{, no update} \\ &\quad \text{will happen in the interval } (t,\ t+\tau), \end{aligned}$$

Now $\mathbb{P}(\tau, \mu)d\tau$ can be expressed as the product of $\mathbb{P}_0(\tau)$ and $a_\mu d\tau$, i.e.

$$\mathbb{P}(\tau, \mu)d\tau = \mathbb{P}_0(\tau) \cdot a_\mu d\tau \tag{1}$$

Since $1 - \sum_i a_i d\tau$ is the probability that no update will happen within $d\tau$ time, the following holds:

$$\mathbb{P}_0(\tau + d\tau) = \mathbb{P}_0(\tau)(1 - \sum_i a_i d\tau)$$

Rewriting this and, since $d\tau$ is infinitesimal, considering the Taylor polynomial around zero, this gives:

$$\mathbb{P}_0(\tau) = e^{-\sum_i a_i \cdot \tau}$$

Finally, substituting this into (1) gives:

$$\mathbb{P}(\tau, \mu) = a_\mu e^{-\sum_i a_i \cdot \tau}$$

Now, taking $r_1, r_2 \sim \mathcal{U}(0,1)$ and setting:

$$\tau = \frac{1}{a_0}\ln(\frac{1}{r_1}) \text{ and } \sum_{v=1}^{\mu-1} a_v < r_2 a_0 < \sum_{v=1}^{\mu} a_v$$

The first rule generates a random value τ according to the density function $P_1(\tau) = a_0 e^{-a_0\tau}$ and the second rule generates a random integer μ according to the density function $P_2(\mu) = \frac{a_\mu}{a_0}$.
Since $P_1(\tau) \cdot P_2(\mu) = P(\tau, \mu)$, a random pair (τ, μ) is generated according to the density functon $P(\tau, \mu) = a_\mu e^{-\sum_i a_i \cdot \tau}$

These two rules will be used in the new algorithm. The general structure of this algorithm is as follows:

1. Create a list of all possible updates: iterate over each element of the array, and, for each element, iterate over each neighbor. If this neighbor differs in type, calculate ΔH and store this in a database.
2. Determine all $a_i = \mathbb{P}(\Delta H) = e^{-\frac{\Delta H}{T}}$ for each i.
3. Sort the records of the database descending on a_i.
4. Determine the sum a_0 of all a_i.
5. Generate two pseudorandom numbers r_1 and r_2 from $[0, 1]$
6. Determine the time until the next update will take place, given by $\tau = \frac{1}{a_0}\ln(\frac{1}{r_1})$, and determine μ such that $\sum_{v=1}^{\mu-1} a_v < r_2 a_0 < \sum_{v=1}^{\mu} a_v$.
7. Increase the time t with τ (this means that τ time passes) and execute the update belonging to a_μ.
8. Renew the database efficiently. This is done by considering the lattice site that changed in the previous step and recalculating ΔH for all possible updates with its neighbors. Since the cell sizes have changed, according to the volume constraint, we also have to change the ΔH values for the updates that consider the lattice sites at the borders of the two cells that have changed in size during the previous step.

9. Calculate the new values for a_i and go back to step 3.

This algorithm is indeed a kinetic Monte Carlo algorithm and works according to our problem. The third step is mainly for debugging and statistics, but can be removed in the final algorithm.

3.2 SQL, SciQL and MonetDB

The implementation of the algortihm is done in C++, but there are places where SQL is needed. SQL is a famous language which is used for Database Management Systems (DBMSs). With this language, queries can be done either to extract information from or to insert information into a database. SciQL is the experimental aspect of this project. This is a modification of SQL specifically meant for grids and arrays to be able to request elements with specific properties in an easy way, developed by M.L. Kersten et al. [8] in 2011 . When the test cases are getting increasingly complicated, we need to handle a very large amount of data (in this project a large amount of possible updates and their characteristics). In theory, SciQL will be very suitable for our problem. Some examples of queries from SciQL and SQL used in this project will be given later on.

The communication of the program with the database with SciQL and SQL is handled by MonetDB. This is software from the Database Architectures group at CWI [9]. MonetDB uses column store technology. This means that a data table is stored as sections of columns of data instead of rows of data. A few advantages of column storing can be noted compared to row storing in terms of efficiency. A row-oriented database reads the whole row to be able to read the desired attribute. Often, queries read much more data than requested. MonetDB is still in full development and keeps improving in terms of functionality, speed and ease of use, partly by feedback from research and applications. Since the integration of SciQL in MonetDB is not entirely finished at this moment, not every function can be used yet, but it is sufficient to use the current version for this project.

There are a few reasons MonetDB and SciQL are chosen to support the kinetic Monte Carlo implementation of the CPM. First of all, it is much easier to use SciQL with an array than straightforward C++ code, since queries can give information in a much more compact way while using less code. Secondly, with MonetDB it is possible to use queries at any desired moment. In this way, debugging and obtaining intermediate statistics can be obtained in a very neat way. Furthermore, with SciQL, it is not possible to go out of the bounds of arrays, so this is not dependent

on any compiler settings. Suppose we want to obtain all types of the neighbors of a certain element that may be on the border of the array. Standard C++ requires the distinction between different cases of the location of this element or the compilation may fail otherwise. With SciQL, if an index is too high, there is simply no record with this index and no information is returned for this value. Finally, in theory, the use of SciQL and MonetDB makes for a very fast program resulting in being able to handle large and complex instances.

An outline of the SQL and SciQL queries that are used in the project follows below [8] [10]. First we consider queries for creating the tables that are needed:

```
CREATE TABLE updates (cellcopiedx INT, cellcopiedy INT,
 celltargetx INT, celltargety INT, energy FLOAT, ai FLOAT,
 cellcopiedtype INT, celltargettype INT);

CREATE ARRAY theworld (x INT DIMENSTION[arrayMAX],
 y INT DIMENSION[arrayMAX], v INT);
```

The first query is a standard SQL in which a table named 'updates' is created. Here, all information of all updates are stored: the indices and type of both the copied element and the target element, the energy associated with the updated (ΔH) and the urge of this update (a_i).
The second query is an example of a SciQL query that creates an array 'theworld'. The dimensions arrayMAX are given for x and y and v represents the value of an element of the array.
The following functions select certain rows from a table.

```
SELECT *
FROM theworld[i][j];
```

This SciQL query is easy to understand. It selects information holding the value of element (i, j) from the array. The following example makes things more complex:

```
SELECT *
FROM theworld[updatetox-1 : updatetox+2][updatetoxy-1 : updatetoy+2]
WHERE v <> updatefromtype;
```

In this query, all neighbors of a certain element with the coordinates 'updatetox' and 'updatetoy' of another type are selected. The <> operator represents the not equal operator. It's important to note that the interval selected has an open endpoint. This means that the indices 'updatetox+2' and 'updatetoy+2' are not included. Thus, the query above is equivalent to the following standard query:

```
SELECT *
FROM theworld
WHERE x BETWEEN updatetox-1 AND updatetox+1
  AND y BETWEEN updatetoy-1 AND updatetoy+1
  AND v <> updatefromtype;
```

Another important ability is to update the database. In the following example, values specified in an external file are being stored in the table 'theworld'.

```
UPDATE theworld
SET v = kar - 48 WHERE x = i AND y = j;
```

More complex queries are used elsewhere to renew updates after a copy has been made, concerning the change in cell sizes. One is presented below, the other three used are equivalent:

```
UPDATE updates
SET energy = energy + compressionresistance*2,
    ai CASE WHEN = energy + compressionresistance*2 > -20
               THEN ai * exp(-1*compressionresistance*2/temperature)
            ELSE exp(20/temperature)
WHERE ((celltargetx  NOT BETWEEN updatetox-1 AND updatetox+1)
    OR (celltargety NOT BETWEEN updatetoy-1 AND updatetoy+1))
  AND cellcopiedtype = type;
```

When using CASE in an update query, it is important to specify all elements or otherwise the remaining elements are given a null pointer, no matter what their previous value was. This must be avoided to prevent segmentation errors and other aggravating issues.

After a copy has been made, it is necessary to add new possible updates to the table, since the grid has changed. This is done by the following query:

```
INSERT INTO updates
VALUES (fromx, fromy, tox, toy, deltaH, a, fromtype, totype);
```

It is possible to concatenate these values to lessen the number of queries.
Since some possible updates disappear after a copy, these rows have to be dropped too:

```
DELETE FROM updates
WHERE celltargetx BETWEEN updatetox-1 AND updatetox+1
  AND celltargety BETWEEN updatetoy-1 AND updatetoy+1);
```

When the program is closed, we also need to remove the tables. This is done by the following queries:

```
DROP TABLE updates;
DROP ARRAY theworld;
```

Some standard SQL queries that have also been featured are the following COUNT and SUM operations. The former counts the number of rows in the table and the latter sums all values of a_i. Both queries return their respective numeric result.

```
SELECT COUNT(*) FROM updates;
SELECT SUM(ai) FROM updates;
```

3.3 Implementation of the new algorithm

The full code of the implementation is given in the appendix. Here, the general structure of the code will be outlined.
There is one object 'Life' in which all relevant functions and variables are present. The rest of the code can be broken down into the following parts:

- Database functions:
  - `query(Mapi dbh, char *q)`
    Extracts information from the database with the query *q in the form of a string.
  - `update(Mapi dbh, char *q)`
    Inserts information to the database with the query *q in the form of a string.
  - `die(Mapi dbh, Mapi hdl)`
    Handles the queries that are given in the previous two functions.
  - `closeTable{Mapi dbh)`
    Drops the tables and handles conflicts upon closing the program.

- General program and settings functions:
  - `menu(Life & life)`
    Main menu for selecting options.
  - `readoption()`
    Reads and returns the option given in the main menu.
  - `input()`
    Reads in an configuration from an external file.
  - `show()`
    Shows the current configuration of the grid in the terminal.
  - `clean()`
    Clears the grid.
  - `showSizes()`
    Gives statistics about the current sizes of the cells.
  - `computeHamiltonian()`
    Computes the Hamiltonian H for statistics and plots
  - `output(double time, Mapi dbh)`
    Generates an textfile of the current configuration for visualisation purposes with Matlab (See the section 'Results').
  - `changeIdealSize()`
    Gives statistics about the current target sizes and the ability to change them.
  - `changeAdhesion()`
    Gives statistics about the current adhesion energies and the ability to change them.
- Algorithmic functions:
  - `fillInUpdates(Mapi dbh)`
    Initializes the table updates when a new configuration has been loaded.
  - `oneStep()`
    Simulates one copy.
  - `multipleSteps()`
    Simulates a given amount of copies.

- `simulateTime()`
  Simulates a given amount of time.

The functions fillInUpdates() and oneStep() will be explained more in detail, because they form the vital part of the program. See the appendix for the full code of these functions.

As stated, fillInUpdates() intializes the table containing the updates. When a configuration has been loaded, first all updates have to be removed from a previous session. After this, the new updates have to be added. The program loops over all elements of the array. For each element with celltype x, the program loops over all its neighbors with celltype other than x. Suppose we have the following situation, and we are looking at element $\boldsymbol{x}$. The neighbors with a different celltype are colored red.

x	x	z	z
x	$\boldsymbol{x}$	y	y
x	y	y	y

These elements are the possible targets for a copy from $\boldsymbol{x}$. Then, for each neighbor element, the change of the Hamiltonian ΔH is calculated. This is done by looping over the elements that are the neighbor of this neighbor. Suppose we're looking at neighbor $\boldsymbol{y}$ directly right to $\boldsymbol{x}$. If the neighbors of the target element of another type are colored blue, we have the following situation before the copy:

x	x	z	z
x	$\boldsymbol{x}$	$\boldsymbol{y}$	y
x	y	y	y

And after the copy:

x	x	z	z
x	$\boldsymbol{x}$	$\boldsymbol{x}$	y
x	y	y	y

Arguing this way, ΔH can be rewritten as follows:

$$\begin{aligned}
\Delta H &= H_{after} - H_{before} \\
&= \sum_{\vec{x},\text{ neighbor of } \boldsymbol{y}} J_{\tau(\sigma(\vec{x})),\tau(x)} \cdot (1 - \delta_{\sigma(\vec{x}),x}) \\
&\quad - \sum_{\vec{x},\text{ neighbor of } \boldsymbol{y}} J_{\tau(\sigma(\vec{x})),\tau(y)} \cdot (1 - \delta_{\sigma(\vec{x}),y}) \\
&\quad + \lambda \Big[(\alpha_x + 1 - A_x)^2 + (\alpha_y - 1 - A_y)^2 \Big] \\
&\quad - \lambda \Big[(\alpha_x - A_x)^2 + (\alpha_y - A_y)^2 \Big] \\
&= \sum_{\vec{x},\text{ neighbor of } \boldsymbol{y}} J_{\tau(\sigma(\vec{x})),\tau(x)} \cdot (1 - \delta_{\sigma(\vec{x}),x}) - J_{\tau(\sigma(\vec{x})),\tau(y)} \cdot (1 - \delta_{\sigma(\vec{x}),y}) \\
&\quad + 2\lambda(\alpha_x - \alpha_y - A_x + A_y + 1)
\end{aligned}$$

The formula above holds when both types are not the medium, i.e. $x \neq 0$ and $y \neq 0$. For $x = 0$ we have:

$$\begin{aligned}
\Delta H = &\sum_{\vec{x},\text{ neighbor of } \boldsymbol{y}} J_{\tau(\sigma(\vec{x})),\tau(0)} \cdot (1 - \delta_{\sigma(\vec{x}),0}) - J_{\tau(\sigma(\vec{x})),\tau(y)}(1 - \delta_{\sigma(\vec{x}),y}) \\
&+ \lambda(-2\alpha_y + 2A_y + 1)
\end{aligned}$$

And for $y = 0$:

$$\begin{aligned}
\Delta H = &\sum_{\vec{x},\text{ neighbor of } \boldsymbol{y}} J_{\tau(\sigma(\vec{x})),\tau(x)} \cdot (1 - \delta_{\sigma(\vec{x}),x}) - J_{\tau(\sigma(\vec{x})),\tau(0)}(1 - \delta_{\sigma(\vec{x}),0}) \\
&+ \lambda(2\alpha_x - 2A_x + 1)
\end{aligned}$$

After computing ΔH this way, the corresponding a-value can be determined. To prevent values from getting too high to be handled by MonetDB, we set, given $T \geq 1$:

$$a_i = \min(e^{-\frac{\Delta H}{T}}, e^{\frac{20}{T}}) \qquad \text{for all } i$$

This rule has another important advantage. By setting an upper limit, the values of a_i are not getting too high, which means that the algorithm will not always take the best copy in terms of energy. When ΔH is very negative, the exponential function increases so much that the other optional copies vanish in terms of probability to be chosen. The values obtained are stored in the table and the loops ensure we get all possible updates.

In the function oneStep(), there first needs to be checked whether the table with updates are empty. If so, the grid is one big cell or empty. In this case, the function should stop to prevent any errors in the rest of the execution since there are no possible copies to be made. Otherwise, the sum of the a-values is calculated. After this, the random numbers $r_1, r_2 \in [0, 1]$ are generated by which τ and μ are being determined according to the formulas given in section 3.1. The information of the update corresponding to μ is being extracted from the table and stored in temporary variables. An update query renews the array according to the copy to be made. This means that the table of updates should be renewed also. Things may get a little tricky here. According to how the Hamiltonian is calculated, all old updates having the celltarget within radius 1 (in terms of a Moore neighborhood) from the element that changed during the copy should be deleted. Suppose the example in the explanation of the function fillInUpdates() above has been executed and $\boldsymbol{x}$ is the new element. Then the updates that have one of the blue elements as a target should be removed:

x	x	z	z	z
x	x	z	z	z
x	x	$\boldsymbol{x}$	y	y
x	y	y	y	y
x	y	y	y	y

When these updates have been removed, new updates have to be added. This is done by looping over all element with radius 2 from $\boldsymbol{x}$. For each element, only the neighbor elements are considered that are within radius 1 of $\boldsymbol{x}$. For all the resulting couples, ΔH and the a-value are calculated and stored in the table in a similar way as in fillInUpdates().
Because of the change in cell sizes, it is important to renew all other updates that are concerned with the same cell types. Suppose a copy has been made from cell-type x to y, like above. Calculating the change of ΔH and a yields the following steps:

For each update i:

- If the x is not the medium, i.e. $x \neq 0$ then:
  - If the target type of the update is x, add 2λ to ΔH_i and $e^{-2\lambda/T}$ to a_i.
  - If the copied type of the update is x, add -2λ to ΔH_i and $e^{2\lambda/T}$ to a_i.
- If the y is not the medium, i.e. $y \neq 0$ then:
  - If the target type of the update is y, add -2λ to ΔH_i and $e^{2\lambda/T}$ to a_i.
  - If the copied type of the update is y, add 2λ to ΔH_i and $e^{-2\lambda/T}$ to a_i.

Of course, we need to keep $a_i \leq e^{\frac{20}{T}}$ for all i. Now, the update table has been renewed correctly and the algorithm can proceed to a new iteration.

During the implementation, some serious problems were encountered. First, using MonetDB in combination with the Mapi library for communication with the server is complicated in Windows. It requires much attention towards the location of the libraries and include files. In the end, the implementation has been made for Linux. For visualization purposes, first OpenGL was considered, but since the system used for this project did not support it, the attention was shifted to a simpler visualization with Matlab.
Another practical problem was the relatively low speed of the program. This centers around the fact two Mapi functions, mapi_query() and mapi_update(), take significantly more computation time than all other functions used. Especially the latter turns down the speed slightly. Thus, these functions needed to be executed as little as possible in critical parts of the program. This means that some optimizations were needed.

The first one concerned the general structure of the code in the two functions fillInUpdates() and oneStep() obtaining information from the database. In the following part of the code, all neighbors of a certain element with the coordinates 'fromx' and 'fromy' are selected:

```
for (int tox = fromx-1; tox <= fromx+1; tox++){
 for (int toy = fromy-1; toy <= fromy+1; toy++){
  if (tox >= 0 && tox < arrayMAX && toy >=0 && toy < arrayMAX &&
  !(tox == fromx && toy == fromy)){
   sprintf(buffer,"SELECT *
                   FROM theworld
                   WHERE x = %d AND y = %d",
                   tox, toy);
   hdl = query(dbh, buffer);
   ...
  }//if
 }//for
}//for
```

By this way of coding, the function query() has to be executed 8 times. This can be toned down by using the following:

```
sprintf(buffer,"SELECT *
                FROM theworld[%d:%d+1][%d][%d+1]
                WHERE v <> %d
                fromx-1, fromx+1, fromy-1, fromy+1, fromtype);
hdl = query(dbh, buffer);
while(mapi_fetch_row(hdl)){
  ...
}//while
```

Now the query() function needs to be executed only one time. In the function oneStep(), this code is part of an large while loop, so it pays in terms of complexity to apply this optimization. Moreover, if more elements need to be selected in case of renewing updates because of changed cell sizes, then the for-loops code have complexity $O(n \cdot m)$, where $n \times m$ is the size of the array. Also, the capabilities of the SciQL language are used here in a much more effective way. The selection of elements is easier to program and the boundary situations do not have to be taken into account.

The second - and more important - optimization is to reduce the number of times that update() is executed. The oneStep() contains a section in which new updates have to be added to the table 'updates' when the copy has been executed. Suppose that a list of neighbors has been generated. With the following code, the corresponding updates can be added to the table:

```
while(mapi_fetch_row(hdl)){
 ... //calculate all needed values here
 sprintf(buffer, "INSERT INTO theworld
                  VALUES (%d, %d, %d, %d, %f, %f, %d, %d)",
                   i, j, updatetox, updatetoy, deltaH, a, fromtype,
                      type);
 update(dbh, buffer);
}//while
```

But again, here we are excuting update() too often. A solution is to concatenate the strings and call update() after the while loop has been completed. Then, the code looks as follows:

```
bool first = true;
int total_buf_size = sprintf( buffer, "INSERT INTO updates VALUES");
while(mapi_fetch_row(hdl)){
 ... //calculate all needed values
 if (first) first = false; //this is needed to avoid an
                           //extra comma at the beginning
 else{
  sprintf(buffer, total_buf_size, ",");
  total_buf_size++;
 }//else
 int len = sprintf(buffer, %d, %d, %d, %d, %f, %f, %d, %d)",
                           i, j, updatetox, updatetoy,
                           deltaH, a, fromtype, type);
 total_buf_size++;
 update(dbh, buffer);
}//while
sprintf(buffer + total_buf_size, ";");
total_buf_size++;
update(dbh, buffer);
```

This optimization has not been applied to the function fillInUpdates() for technical reasons (the string might become too long). This is not much of an issue, since this function will be called only one time upon executing the algorithm.

To give an indication of how well this works out, for a simple example the running time of the function oneStep() has been reduced by a factor of one hundred. Still, for complex examples this could increase until a second. The code could be improved further, but for this project it is currently fast enough to conduct experiments with.

4 Results

To validate the implementation, three experiments are conducted.

4.1 Experiment 1 - One long cell

In the first experiment, the behavior of one long cell will be analyzed. The initial configuration is given below. The indices of the grid are given on the axes. The blue cell has id 1 and the yellow medium has id 0. Here, we have $\alpha_1 = 40$ and take $J_{01} = 1$.

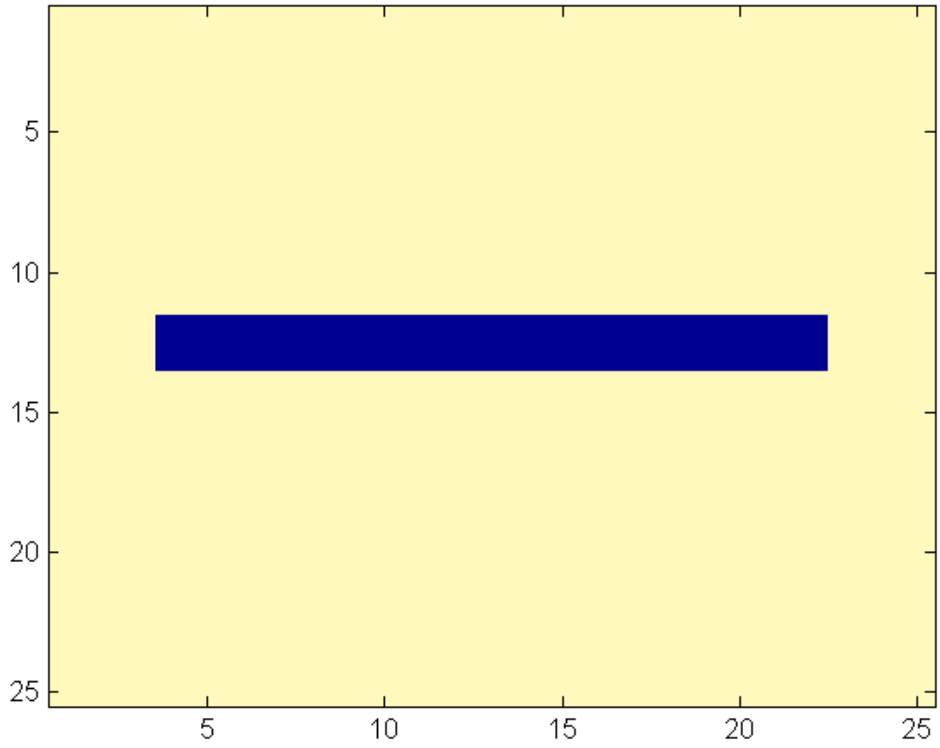

Figure 1: Initial configuration of the grid in experiment 1

4.1.1 Experiment 1.1 - One long cell that shrinks

First, we concentrate on the behavior when the cell shrinks from size 40 to approximately size 15. Thus, we set $A_1 = 15$. The parameter λ takes the values $2, 5, 10$ respectively. For each of these values, T takes the values $1, 5, 10$ and the shapes after three seconds of simulation are given, along with the Hamiltonian H during the first three seconds. The t-axis is given in both linear and logarithmic scale to give an idea when updates take place and how the Hamiltonian will change over time.

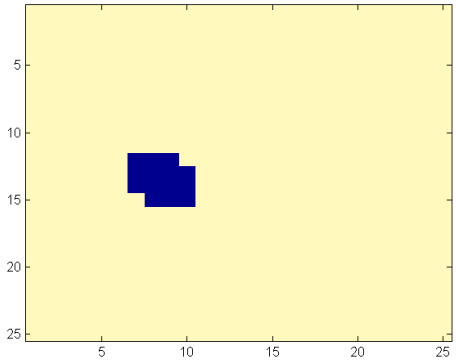

Figure 2: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 1$

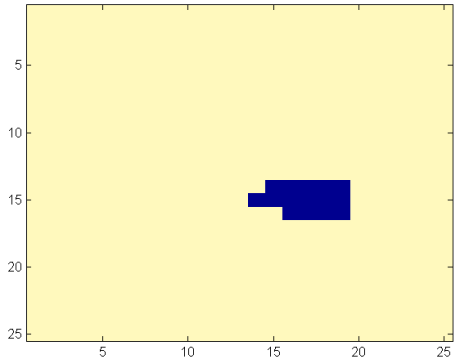

Figure 3: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 5$

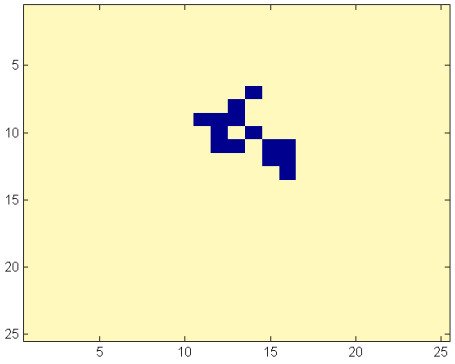

Figure 4: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 10$

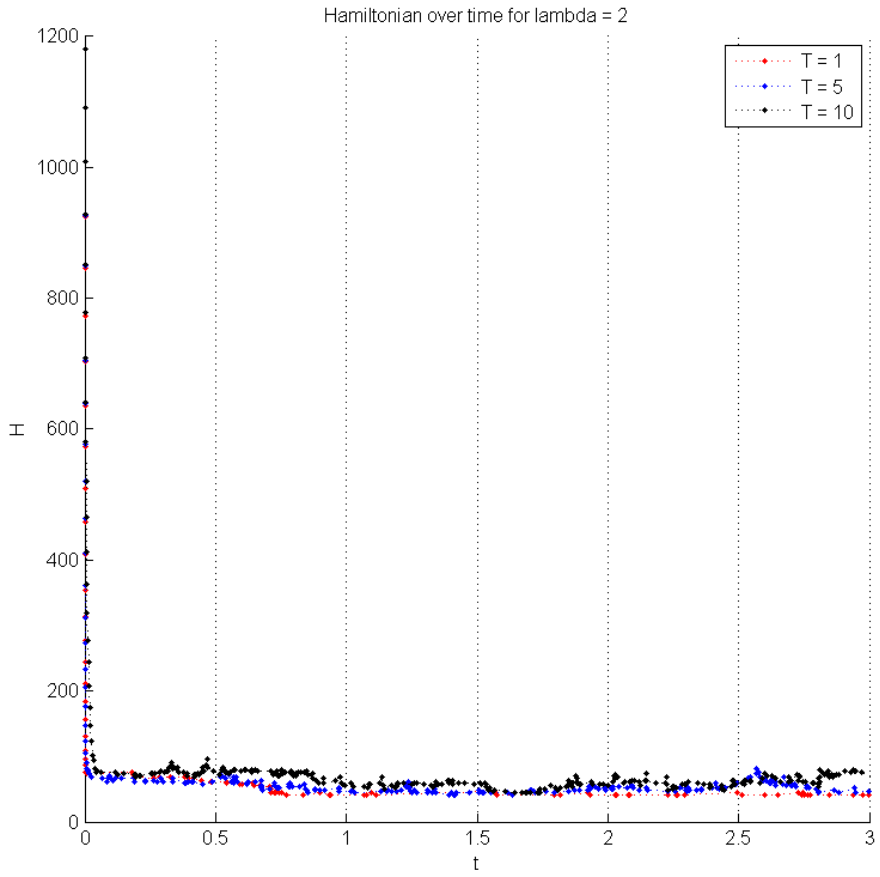

Figure 5: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a linear scale.

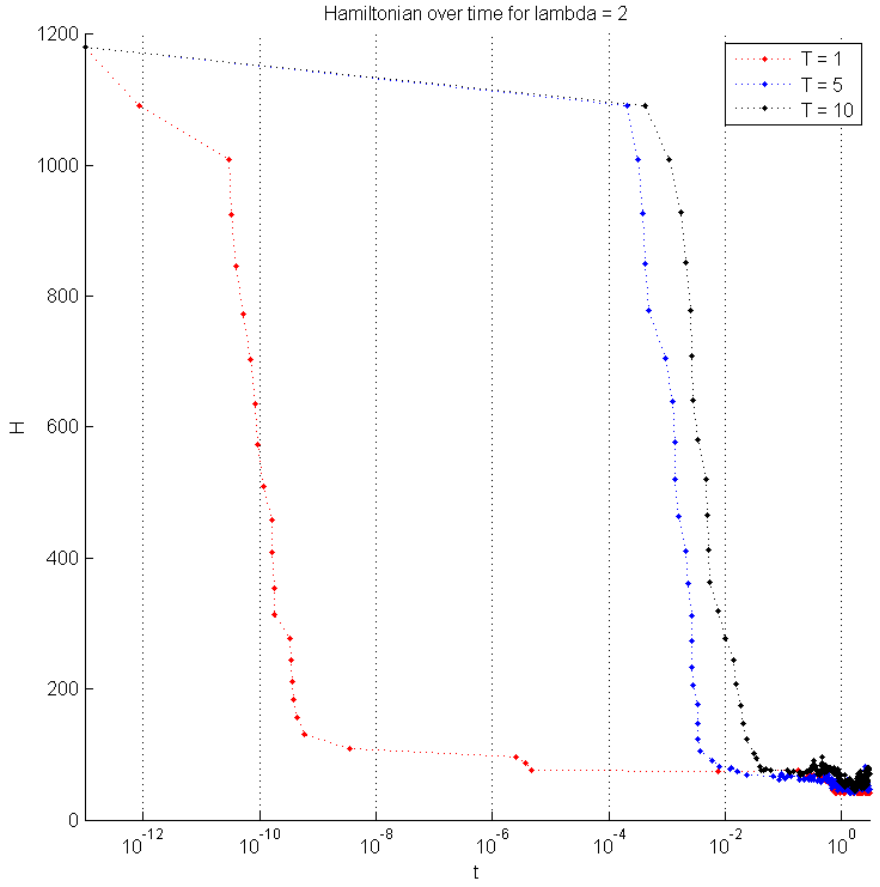

Figure 6: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a logarithmic scale.

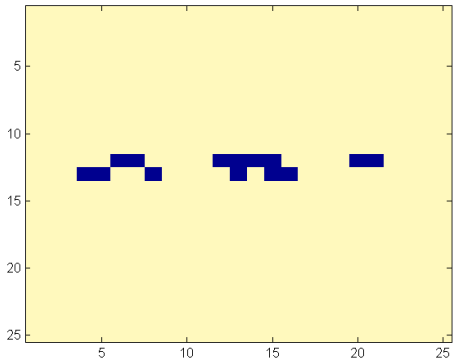

Figure 7: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 1$

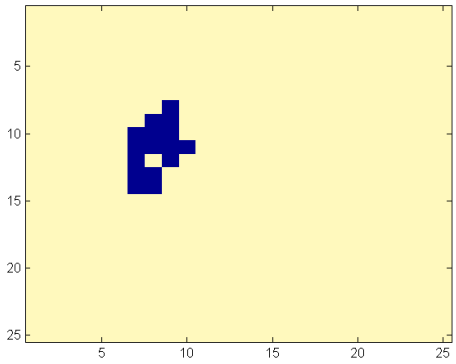

Figure 8: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 5$

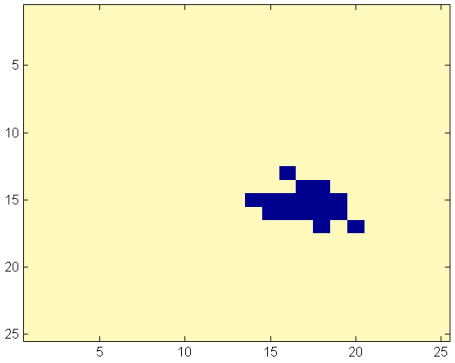

Figure 9: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 10$

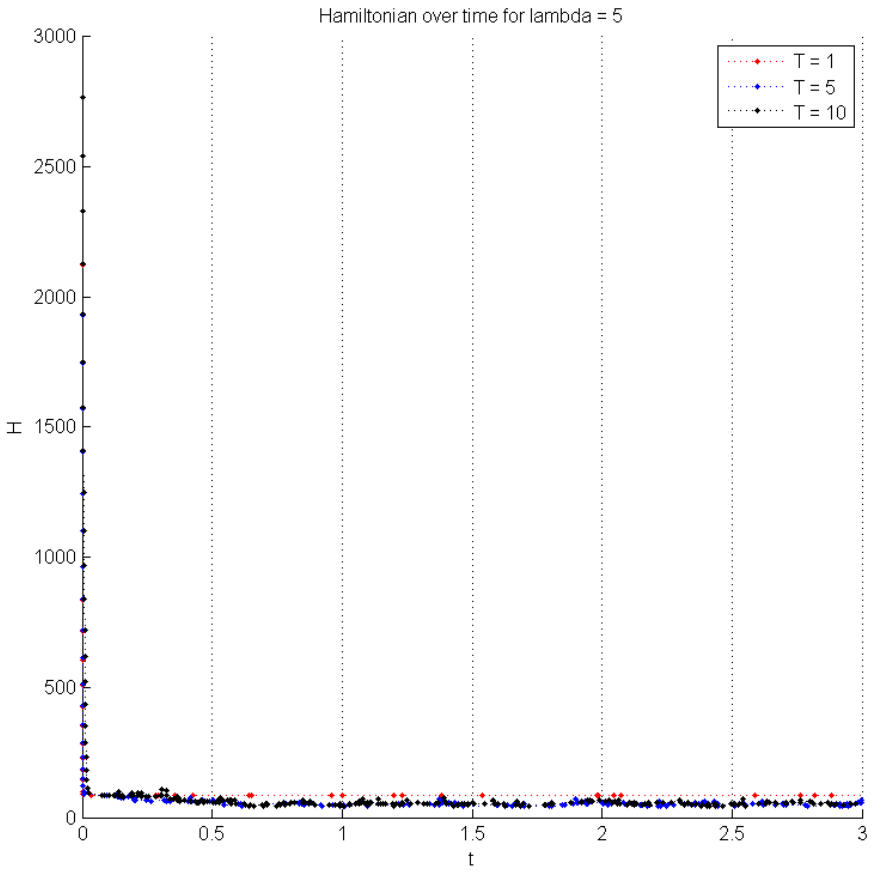

Figure 10: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a linear scale.

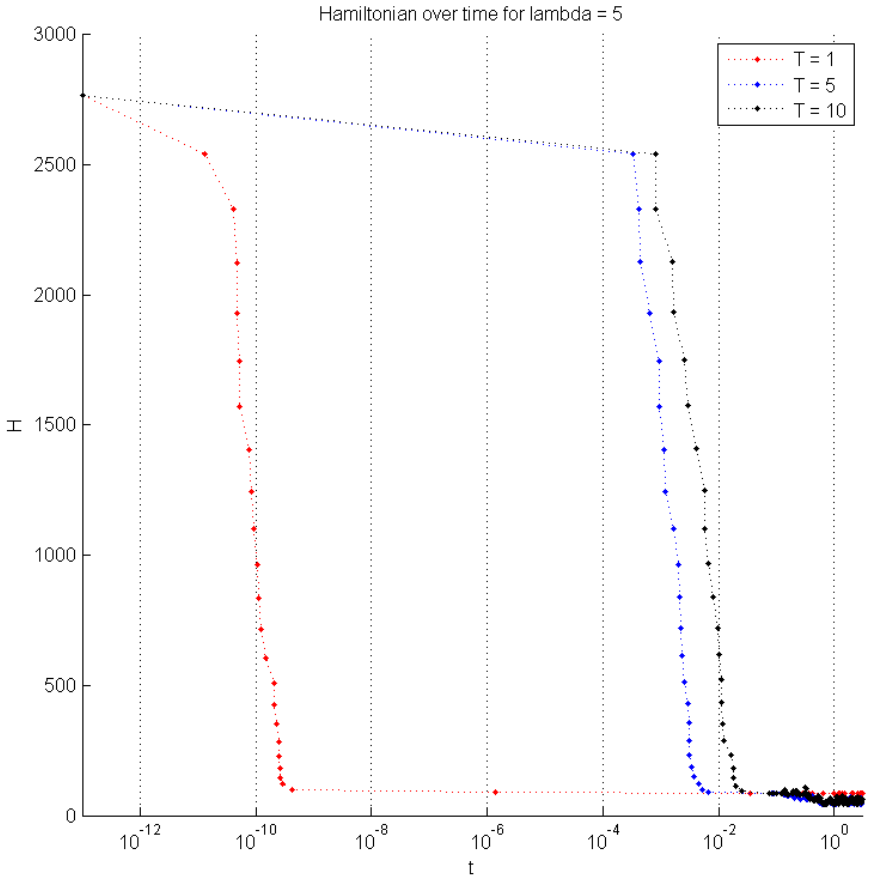

Figure 11: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a logarithmic scale.

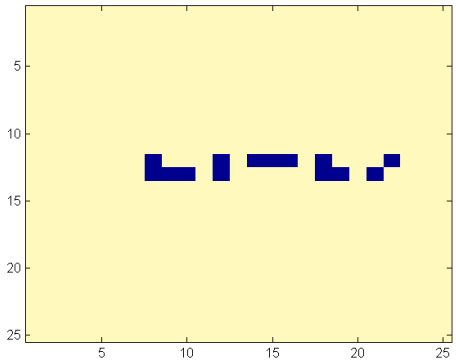

Figure 12: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 1$

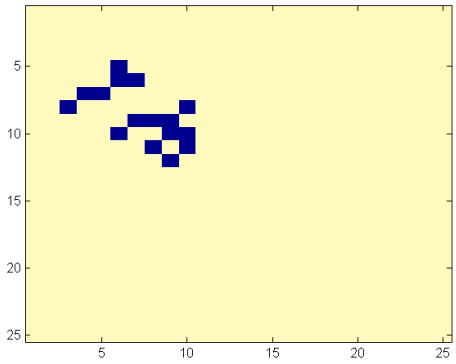

Figure 13: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 5$

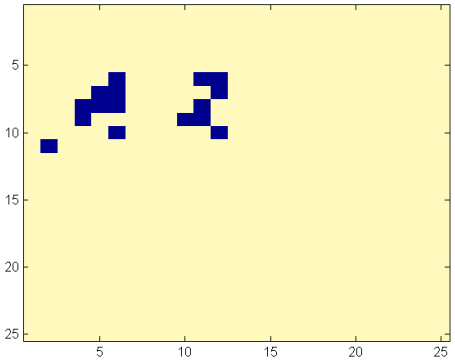

Figure 14: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 10$

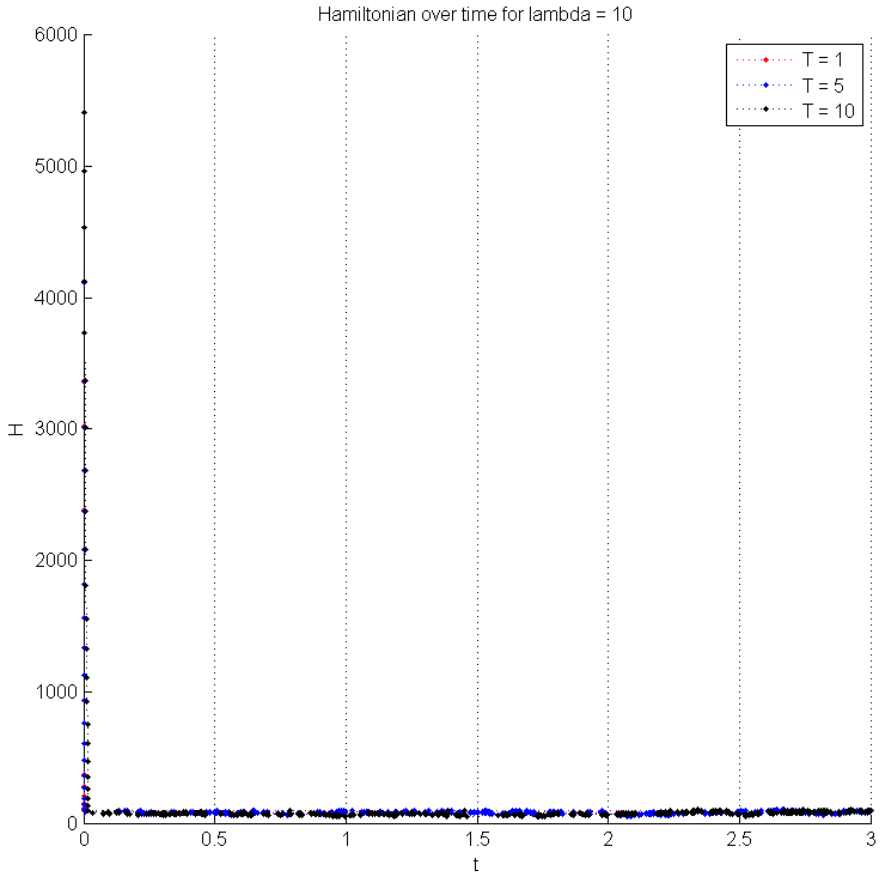

Figure 15: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a linear scale.

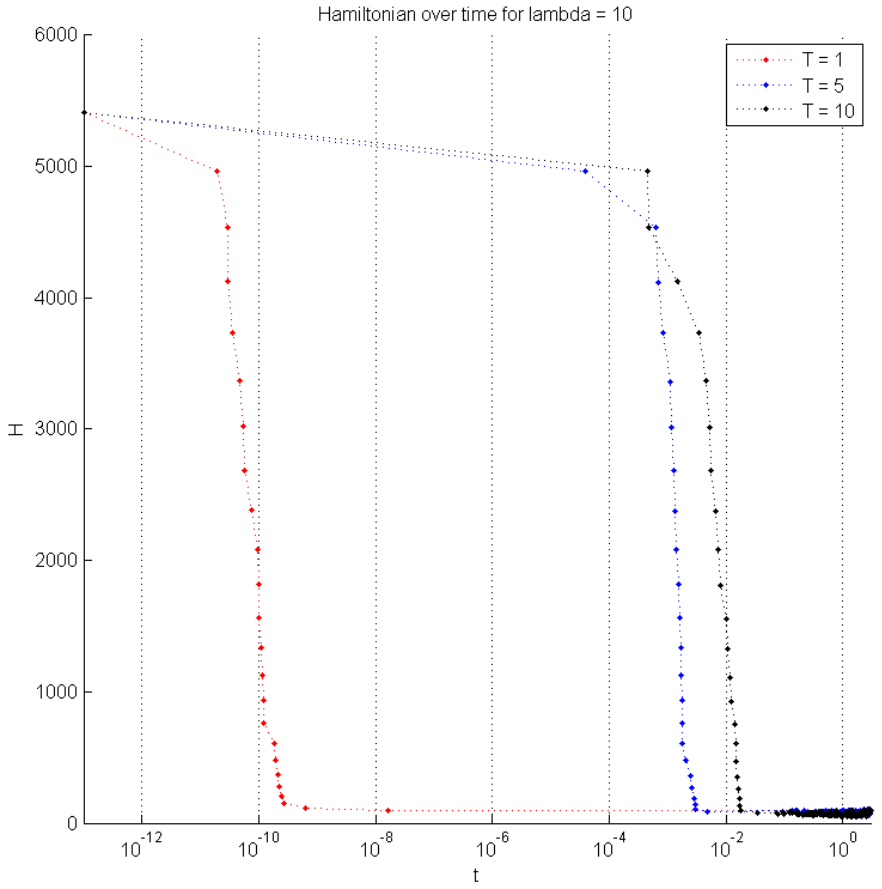

Figure 16: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a logarithmic scale.

Based upon the final configurations, a few things can be noted. First of all, the simulation with $\lambda = 2$ and $T = 1$ gives the most desired result. Here, the cell is compact and round and therefore has the lowest Hamiltonian (which turns out to be 42, which is the theoretical minimum for this example). However, considering how much the Hamiltonian decreases in the plots, the other temperatures yield Hamiltonians that are also not far off in the end. Looking at the configurations, the cell is less round and has more inlets at higher temperatures. Nevertheless, an higher temperature causes the Hamiltonian to drop much later than a lower temperature, looking at the logarithmically scaled plots.

It seems that a higher λ causes the cell to break into multiple parts. For $\lambda = 5$, this happens more when the temperature is low, because then the original shape of the cell stays intact, albeit with many gaps. When λ is high enough, this occurs also at higher temperatures. λ seems to have no effect on when the Hamiltonian drops, but the Hamiltonian is generally a little higher in the end.

It might be confusing that at higher values for λ the cell is less cohesive. This could be explained by the fact that the cell is shrinking and that when λ is high, the initial penalty in terms of the volume constraint is higher. Because of this, the cell shrinks in a rougher way and therefore breaks into multiple parts. When the temperature increases, the parts have more chance to reunite again.

4.1.2 Experiment 1.2 - One long cell that keeps cell size

It would also be interesting to see what the behavior of the cell would be if the cell size stays the same. Thus, we set $A_1 = 40$. Again, we use $J_{01} = 1$ and the parameter λ takes the values 2, 5, 10 respectively. For each of these values, the shape after three seconds of simulation are given, along with the Hamiltonian H during the first three seconds. The t-axis is given in both linear and logarithmic scale to give an idea when updates take place and how the Hamiltonian will change over time. The initial cell configuration is the same as with the previous experiment.

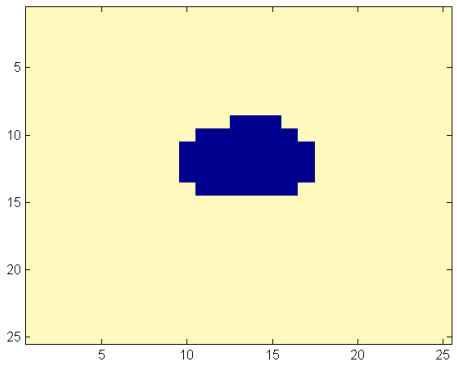

Figure 17: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 1$

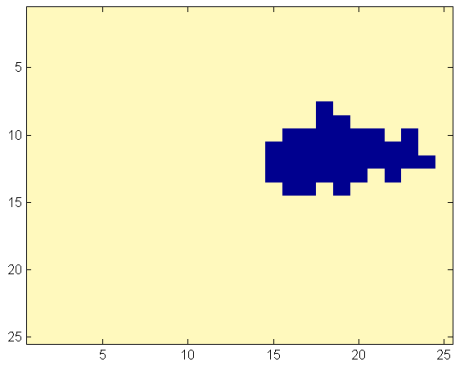

Figure 18: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 5$

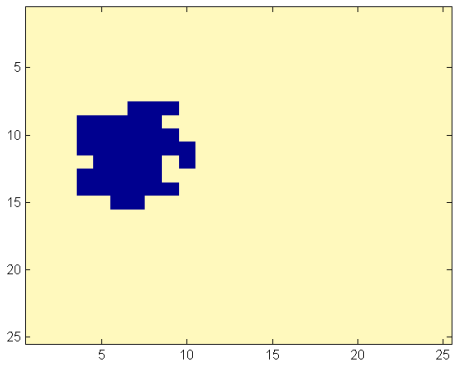

Figure 19: Configuration after 3 seconds of simulation with $\lambda = 2$ and $T = 10$

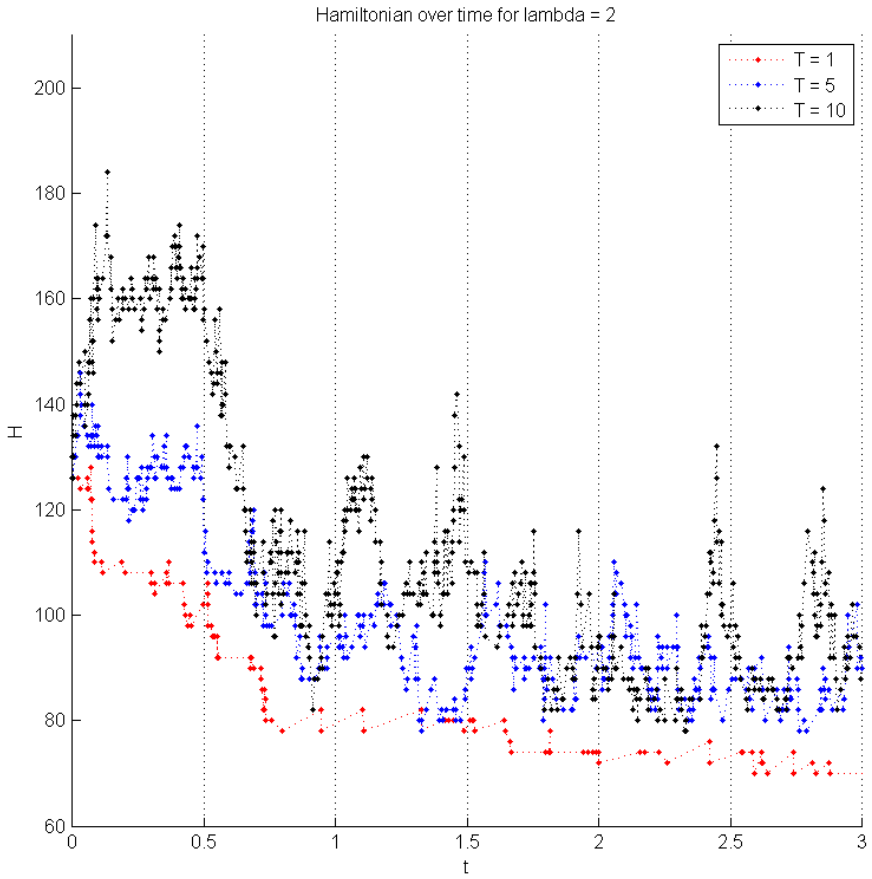

Figure 20: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a linear scale.

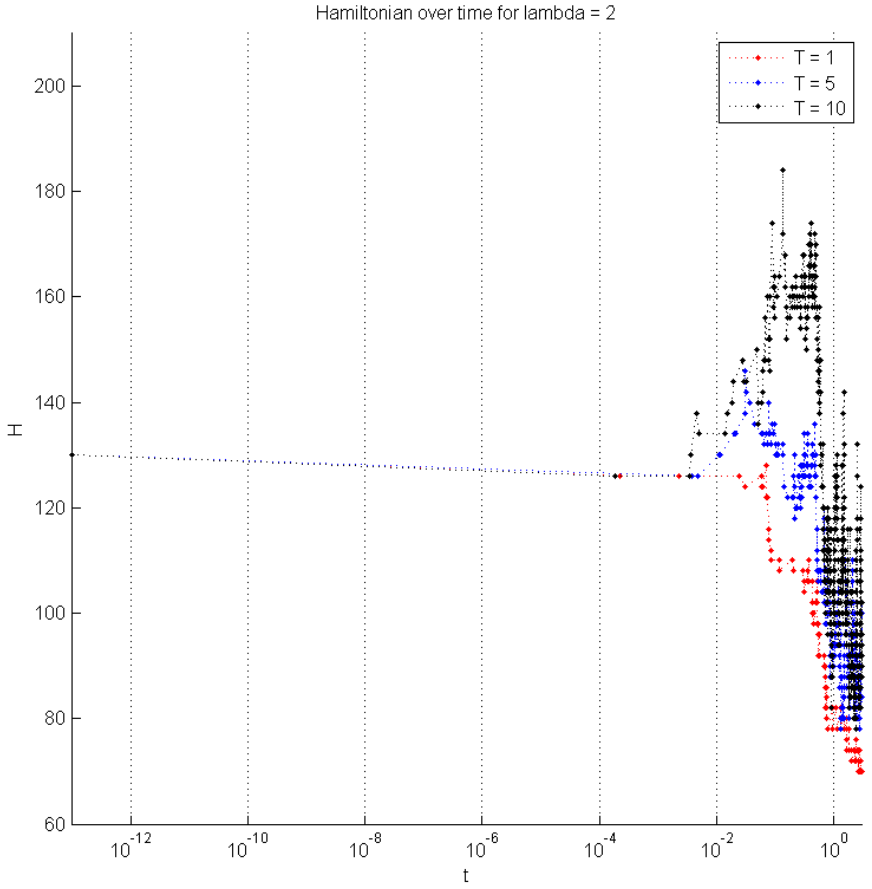

Figure 21: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a logarithmic scale.

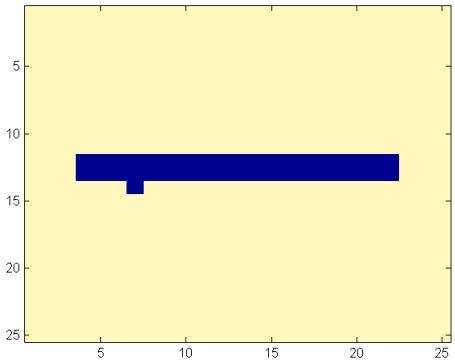

Figure 22: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 1$

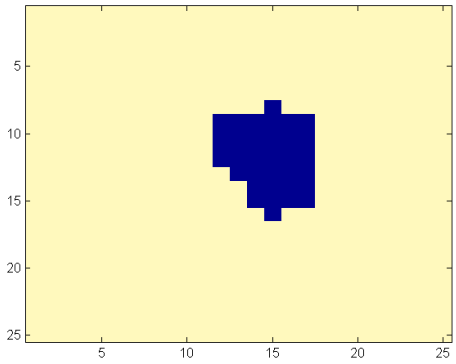

Figure 23: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 5$

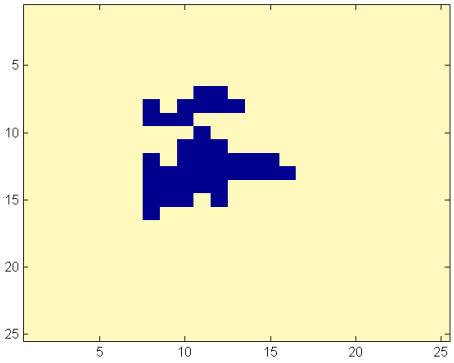

Figure 24: Configuration after 3 seconds of simulation with $\lambda = 5$ and $T = 10$

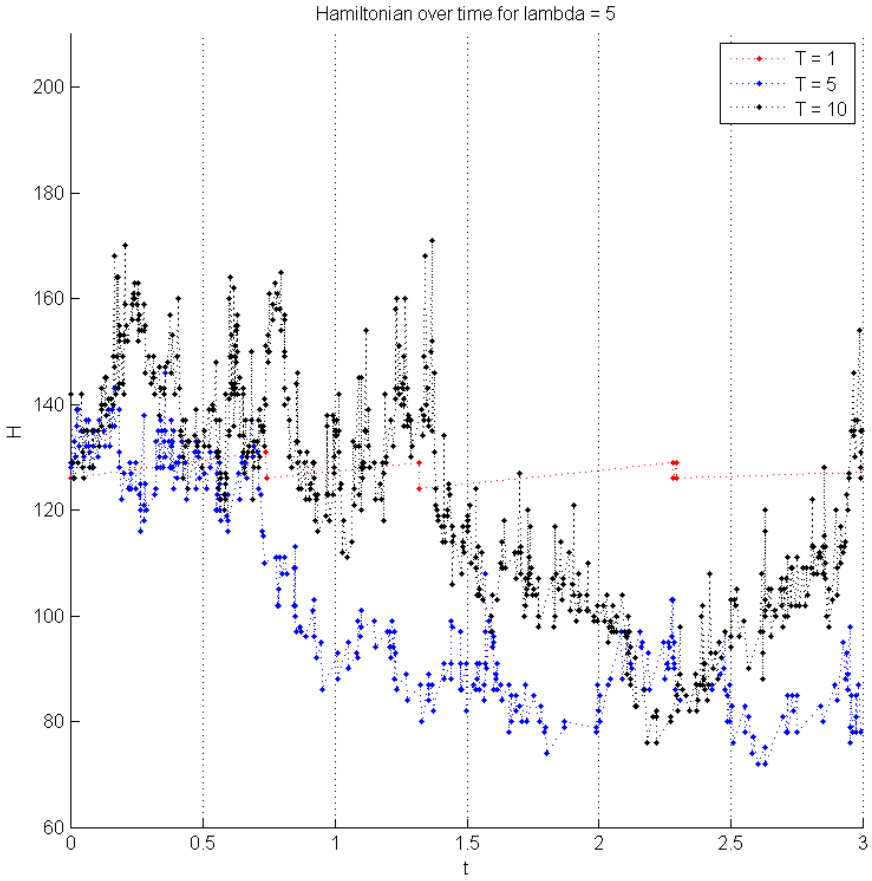

Figure 25: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a linear scale.

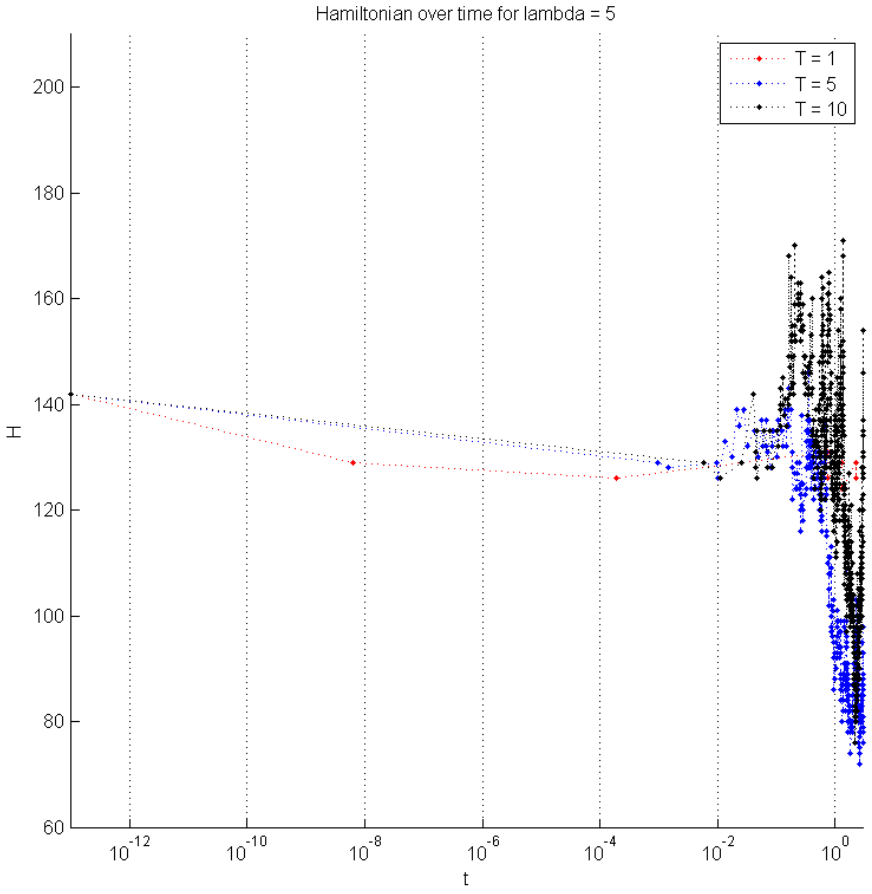

Figure 26: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a logarithmic scale.

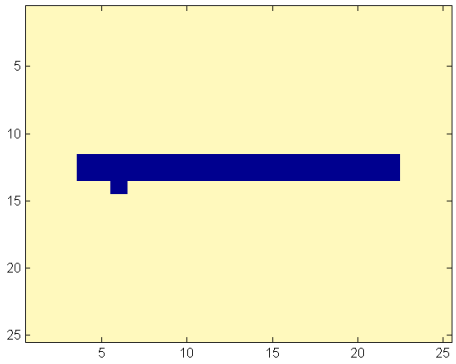

Figure 27: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 1$

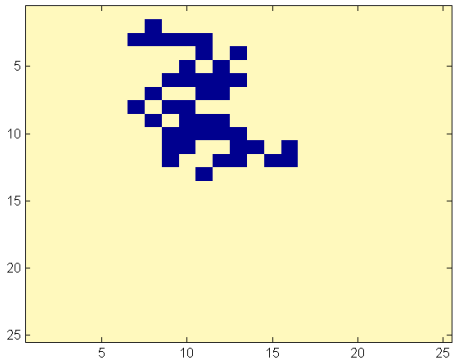

Figure 28: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 5$

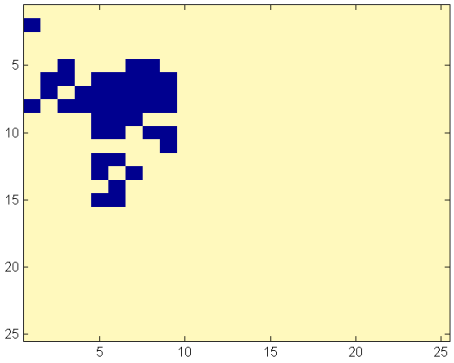

Figure 29: Configuration after 3 seconds of simulation with $\lambda = 10$ and $T = 10$

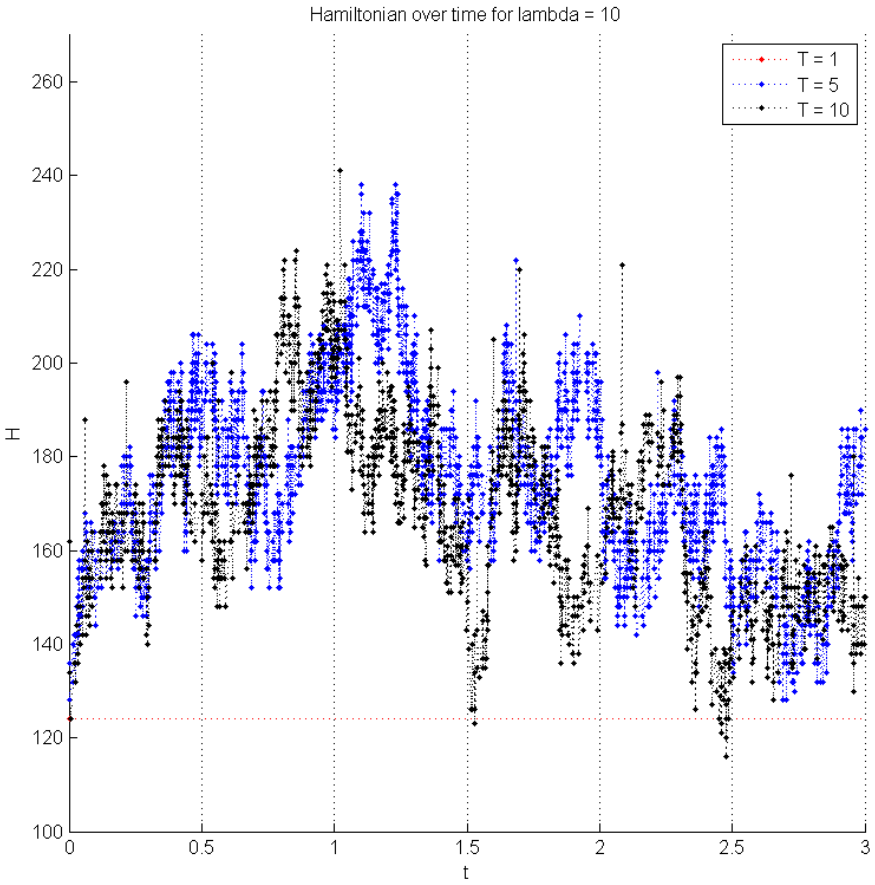

Figure 30: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a linear scale.

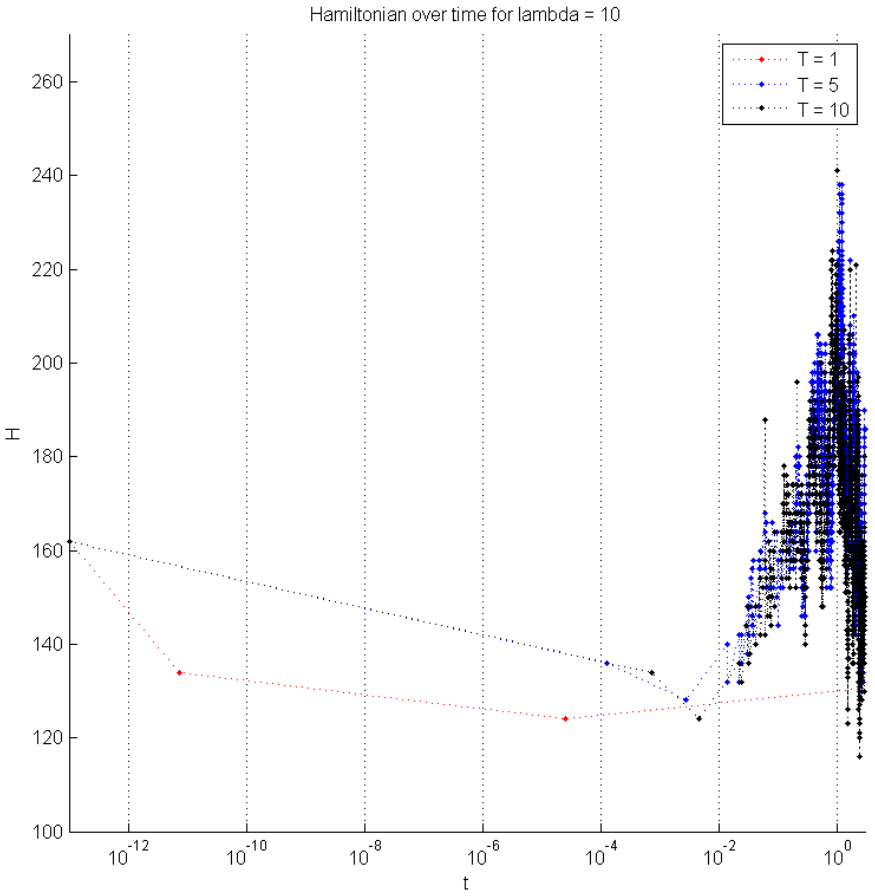

Figure 31: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a logarithmic scale.

Again, the settings $\lambda = 2$ and $T = 1$ give the most desired result in this experiment. The cell has a round shape without inlets. When the temperature increases for $\lambda = 2$, the number of inlets start to increase and the cell loses its round shape. This also means that the Hamiltonian should be higher, which can be seen in the plots. The Hamiltonian is generally higher and fluctuates more when the temperature is high. This also goes for $\lambda = 5$, but note that it takes much time before a new update happens when $T = 1$. Taking a look at the shape, for $T = 1$, the cell almost keeps its original shape, hence the nearly horizontal red line. For both $T = 5$ and $T = 10$, we see similar behavior as for $\lambda = 2$. The cell is round for $T = 5$ and gets increasingly more inlets when $T = 10$. The Hamiltonian has the same behavior as when $\lambda = 2$. For $\lambda = 10$, there is similar behavior. The main difference here is that for $T = 5$ and $T = 10$ the cell has much more inlets. Another notable fact is that for $T = 10$ the Hamiltonian is occasionally much lower than for $T = 5$. This could lie in the fact that the cell might stick to the boundary during the simulation for $T = 10$.

4.2 Experiment 2 - Four similar cells

The next experiment is about four cells of the same type. The initial configuration is given below. The four cells have ids 1 to 4 and the yellow medium has id 0. The four cells have an initial cell size of 36 and a target size of 15. This means $\alpha_i = 36$ and $A_i = 15$ for $1 \leq i \leq 4$. We set $J_{0i} = J_{i0} = 8$ and $J_{ij} = 2$ for $i, j \in \{1, 2, 3, 4\}$. This is done because the cells should be sticking together, because the adhesion energy between the cells and the medium is now much higher than their mutual adhesion. The parameter λ takes the values $2, 5, 10$ respectively. For each of these values, the shape after two seconds of simulation are given, along with the Hamiltonian H during the first two seconds. The t-axis is given in both linear and logarithmic scale to give an idea when updates take place and how the Hamiltonian will change over time.

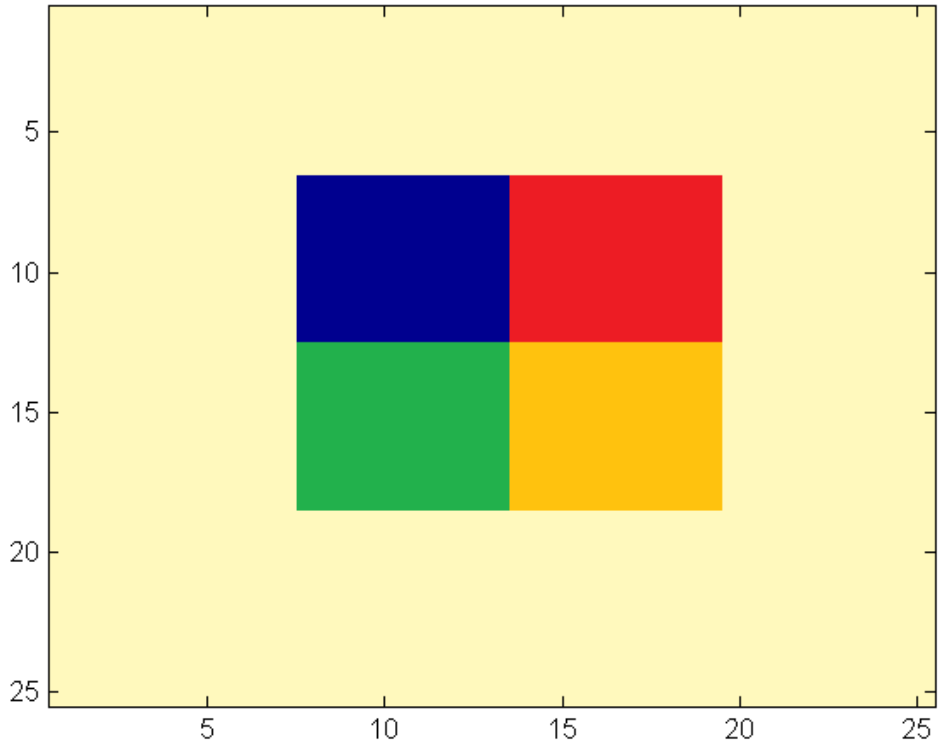

Figure 32: Initial configuration of the grid in experiment 2

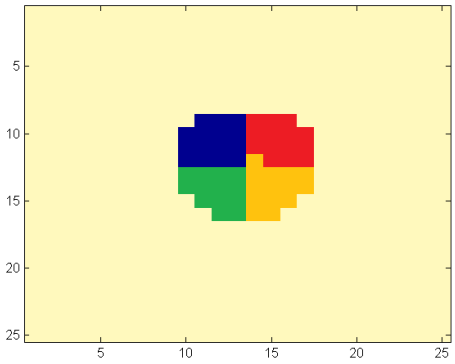

Figure 33: Configuration after 2 seconds of simulation with $\lambda = 2$ and $T = 1$

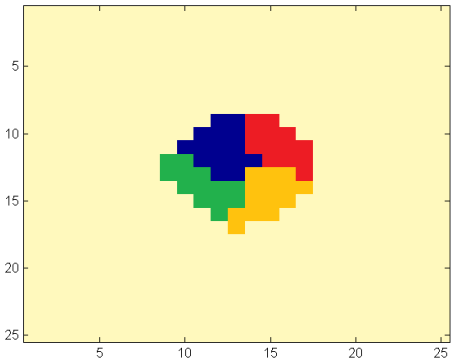

Figure 34: Configuration after 2 seconds of simulation with $\lambda = 2$ and $T = 5$

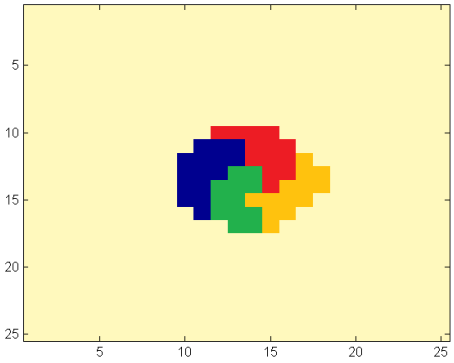

Figure 35: Configuration after 2 seconds of simulation with $\lambda = 2$ and $T = 10$

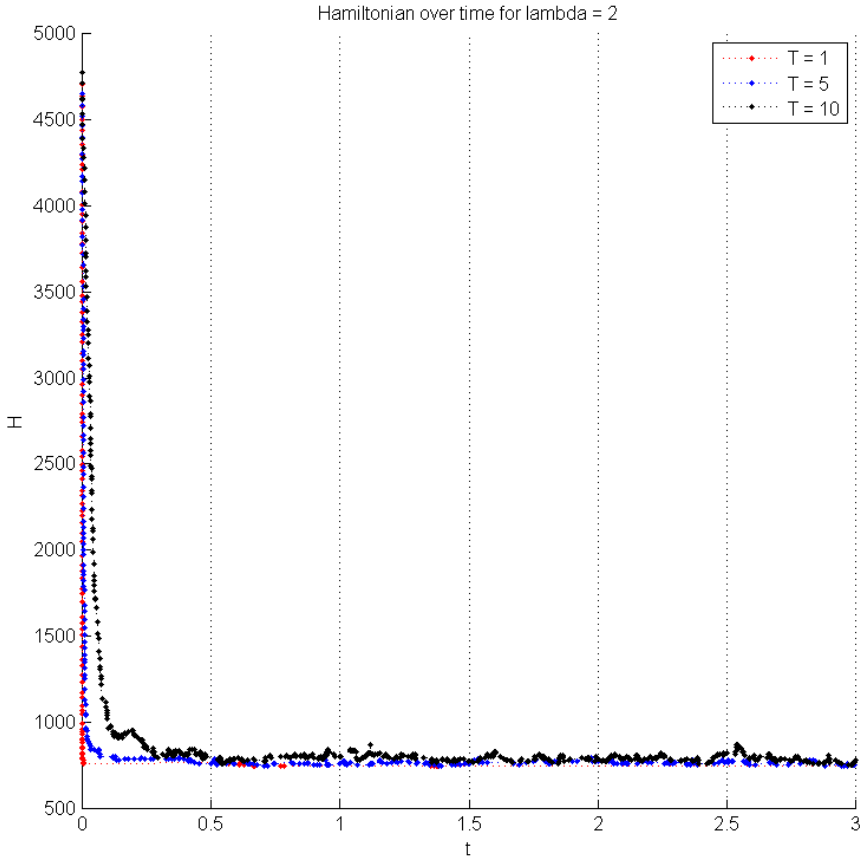

Figure 36: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a linear scale.

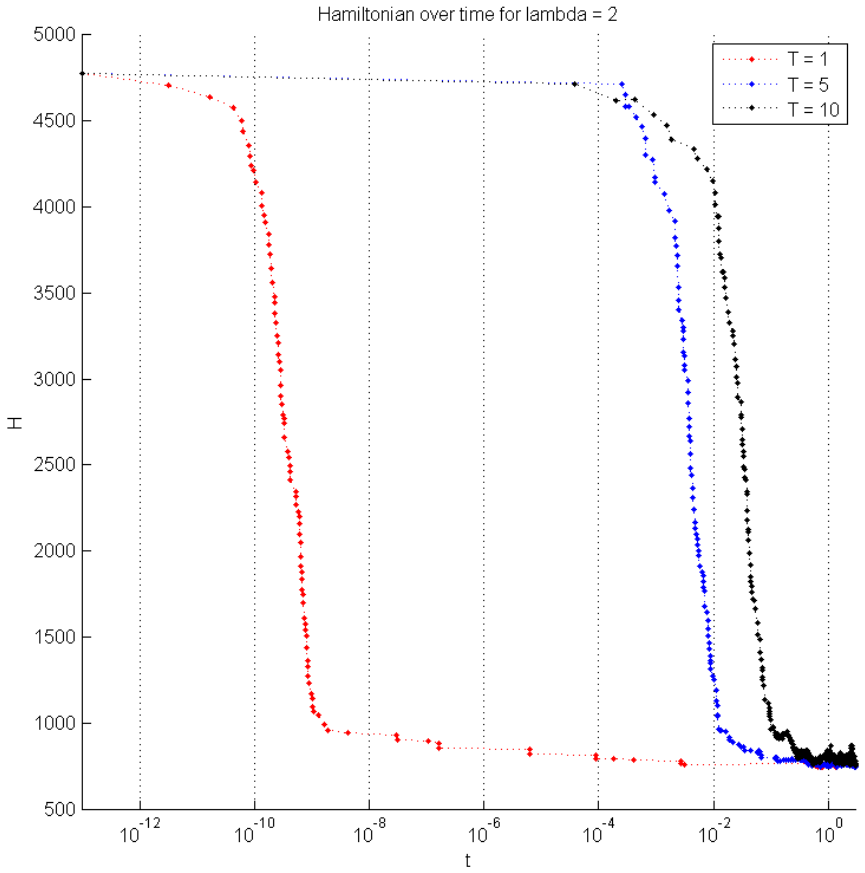

Figure 37: Plot of the Hamiltonian over time with $\lambda = 2$. The t-axis has a logarithmic scale.

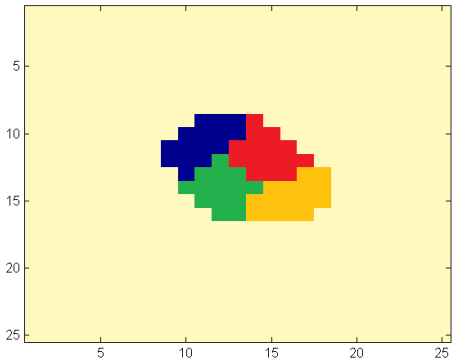

Figure 38: Configuration after 2 seconds of simulation with $\lambda = 5$ and $T = 1$

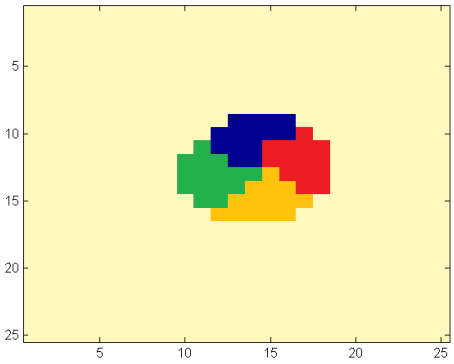

Figure 39: Configuration after 2 seconds of simulation with $\lambda = 5$ and $T = 5$

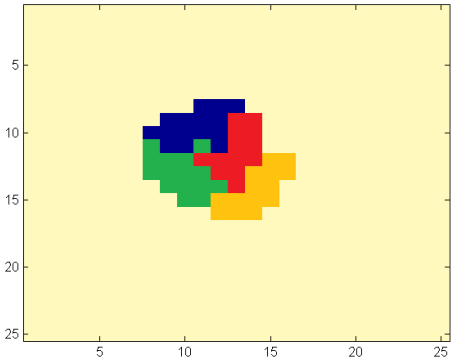

Figure 40: Configuration after 2 seconds of simulation with $\lambda = 5$ and $T = 10$

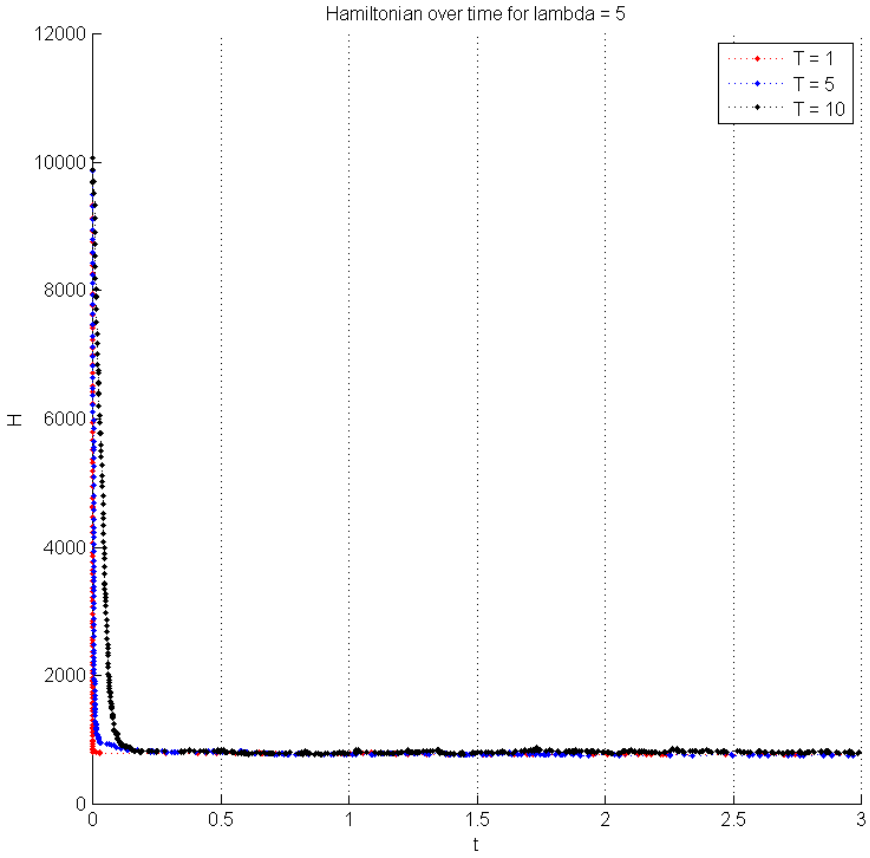

Figure 41: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a linear scale.

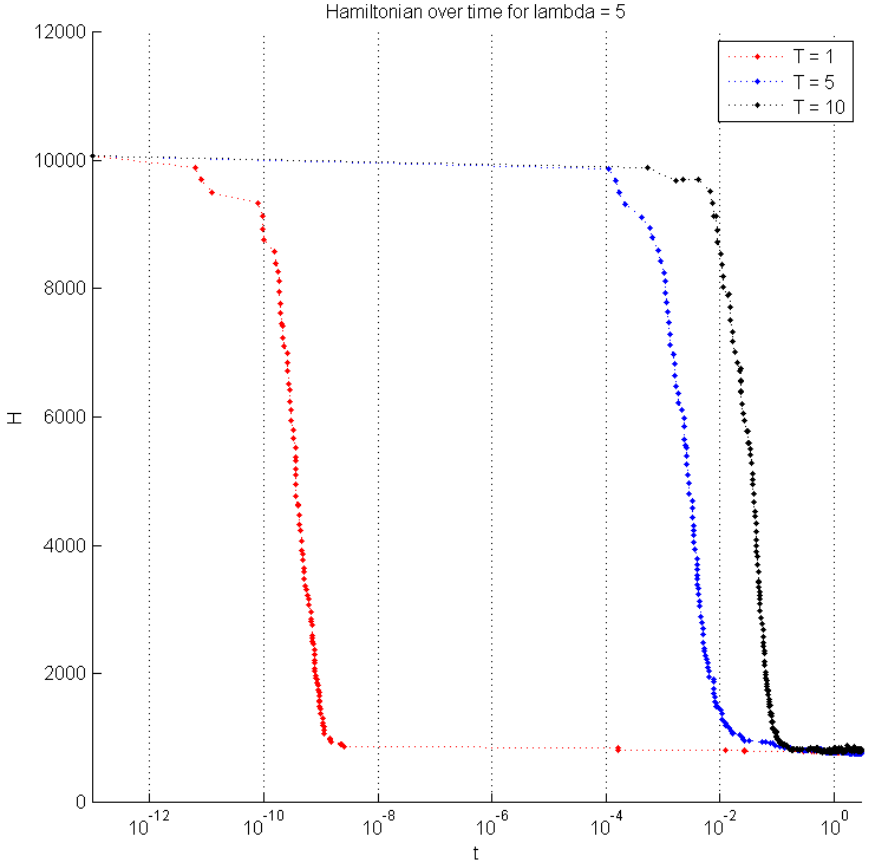

Figure 42: Plot of the Hamiltonian over time with $\lambda = 5$. The t-axis has a logarithmic scale.

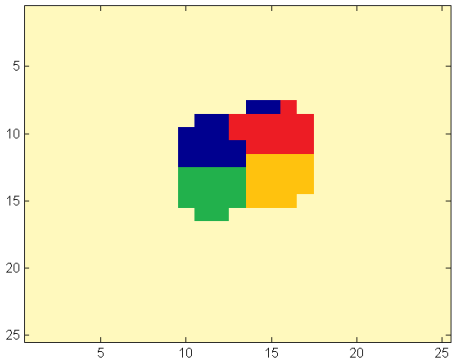

Figure 43: Configuration after 2 seconds of simulation with $\lambda = 10$ and $T = 1$

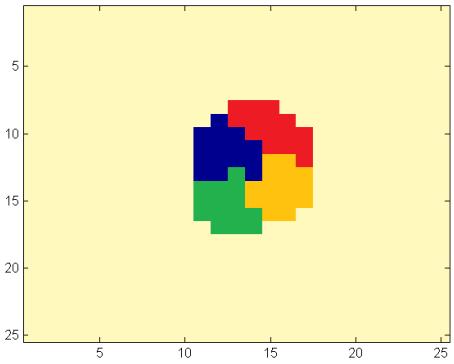

Figure 44: Configuration after 2 seconds of simulation with $\lambda = 10$ and $T = 5$

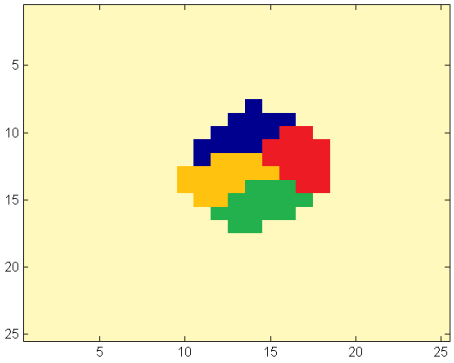

Figure 45: Configuration after 2 seconds of simulation with $\lambda = 10$ and $T = 10$

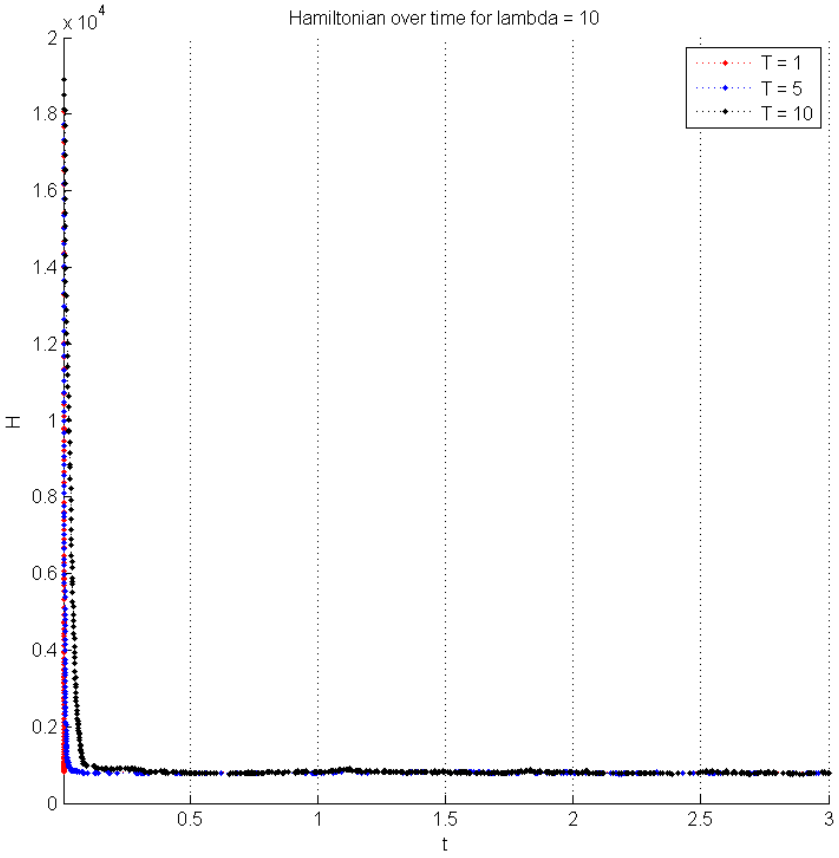

Figure 46: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a linear scale.

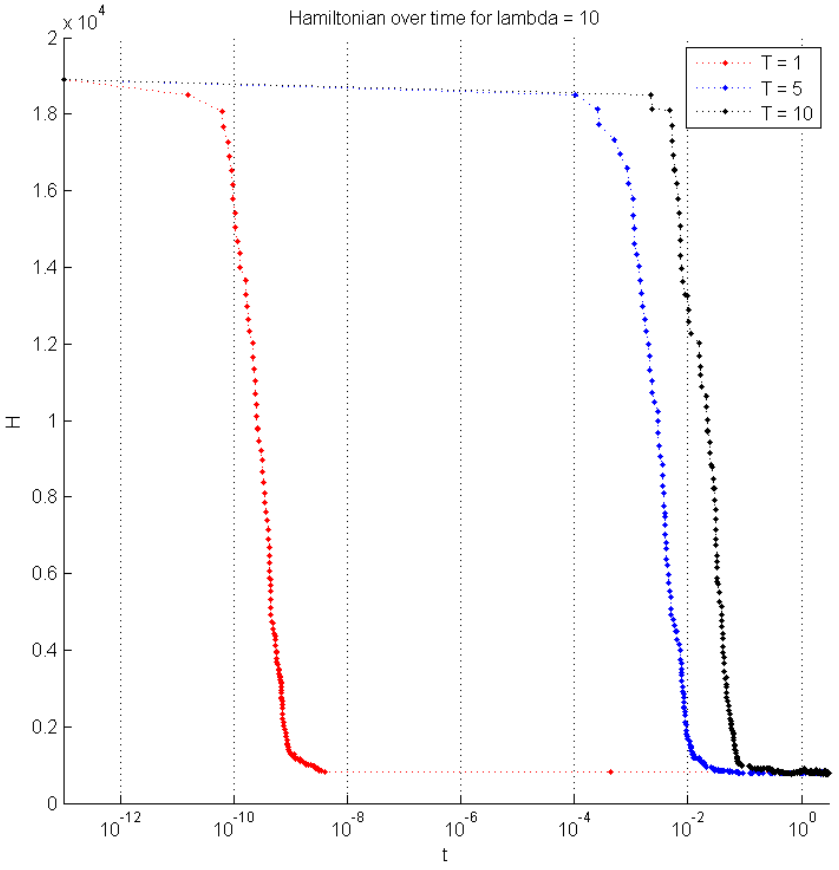

Figure 47: Plot of the Hamiltonian over time with $\lambda = 10$. The t-axis has a logarithmic scale.

In this experiment the cells shrink to the desired size and stick together in each case. For $\lambda = 2$, it is clear that the Hamiltonian is higher and fluctuates more for $T = 10$ then for the other temperatures. For the other values of λ the difference between the Hamiltonians are so relatively small that not much can be said. The values of the Hamiltonian also drop first for $T = 1$, then for $T = 5$ and finally for $T = 10$ and the times at which this happens is the same for each λ. For a higher temperature, the cells lose their round shape a little, but the round shape of the cluster stays. At higher temperatures, the clusters may rotate around the center of the cluster. However, the position of the center of the cluster stays largely the same in every case.

4.3 Experiment 3 - Engulfment of cells by other cells

The final - and most challenging - experiment is inspired by the simulation of the engulfment described by Glazier and Graner. Here, we will use a smaller grid, smaller cells and fewer cells. The initial configuration is given on the next page. The eleven blue cells have ids 1 to 11, the seven red cells have ids 12 to 18 and the yellow medium has id 0. We will mainly use the same parameters:

$$J_{ij} = J_{ji} = \begin{cases} 14 & \text{when } 1 \leq i \leq 11 \text{ and } 1 \leq j \leq 11, \\ 2 & \text{when } 1 \leq i \leq 11 \text{ and } 12 \leq j \leq 18, \\ 11 & \text{when } 12 \leq i \leq 18 \text{ and } 12 \leq j \leq 18, \\ 16 & \text{when } i = 0. \end{cases}$$

$$A_i = 15 \text{ for } 1 \leq i \leq 18$$

It is important for obtaining engulfment to set the red-blue adhesion energy very low compared to all other adhesion energies. In this way, engulfment ensures minimal adhesion energy. Also, the red-medium and the blue-medium adhesion should be set high to keep the cells sticking together. For the other parameters, we take $\lambda = 1$ and $T = 10$. The initial cell sizes have been chosen in a way that they are about 15 in average. The shapes after 1000 and 2000 copies is given, along with the Hamiltonian H during the first 2000 copies. The t-axis is given in a linear scale.

Taking a look at the results, the same phenomenon occurs here. The blue cells start to form a ring around the red cells. The plot shows that the Hamiltonian is decreasing over time. To get certainty whether the simulation follows this trend, the program should have to simulate 5000 or 10000 copies, which is currently too big to handle. But despite that a small grid is used here, the behavior of the cells is very promising.

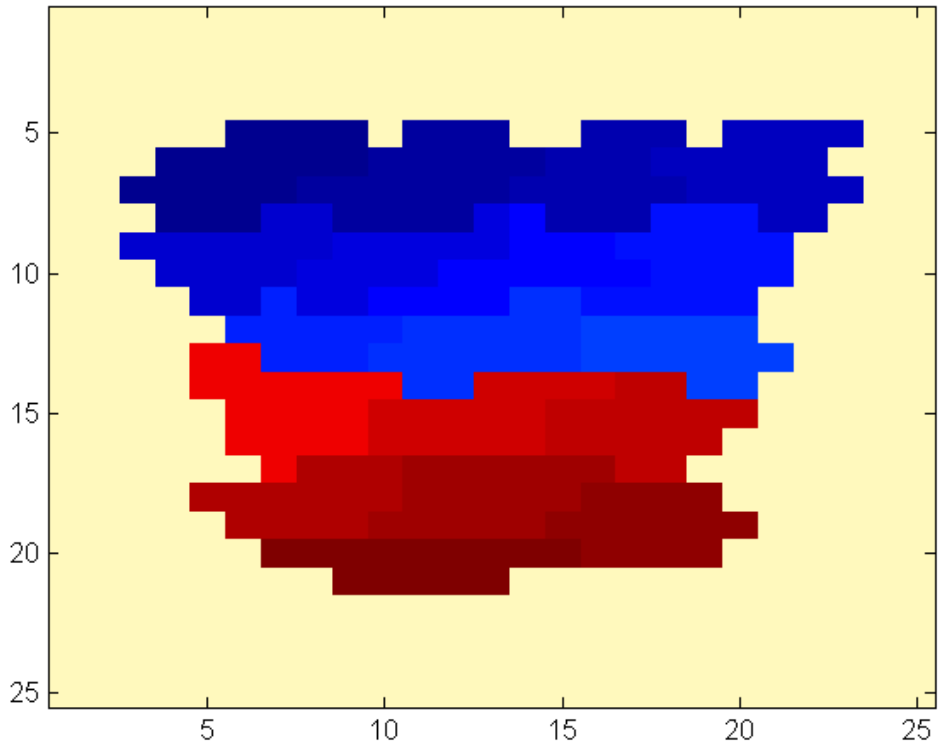

Figure 48: Initial configuration of the grid in experiment 3

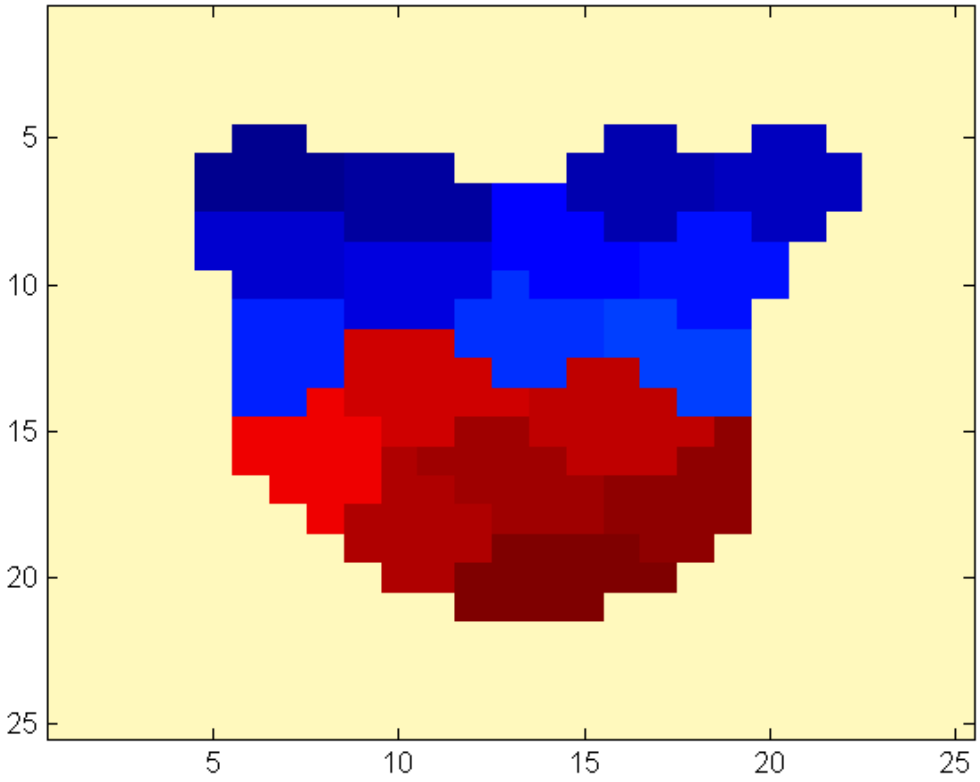

Figure 49: Configuration of the grid in experiment 3 after 1000 copies (approximately 4.17 seconds)

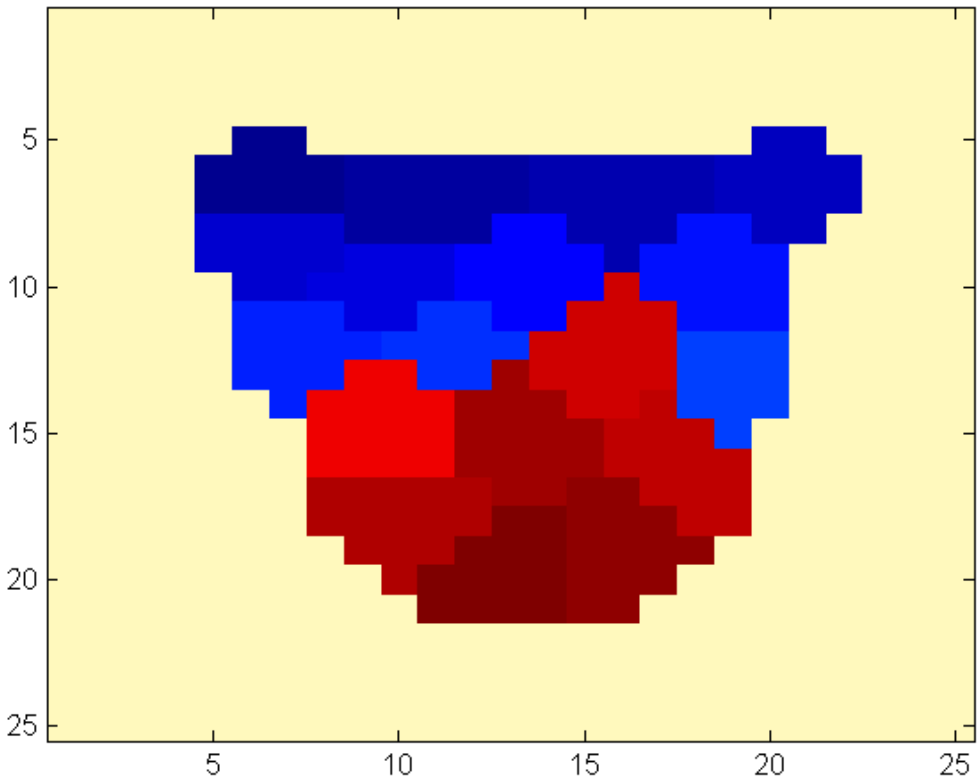

Figure 50: Configuration of the grid in experiment 3 after 2000 copies (approximately 129.06 seconds)

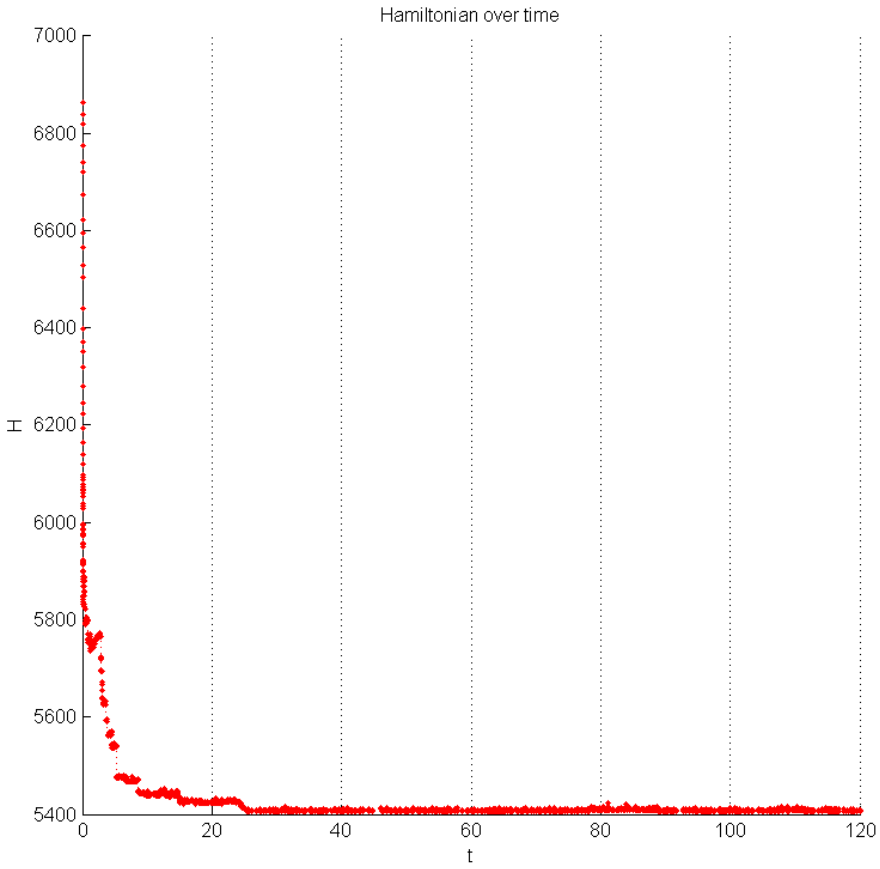

Figure 51: Plot of the Hamiltonian over time for 2000 copies with $\lambda = 1$ and $T = 10$. The t-axis has a linear scale.

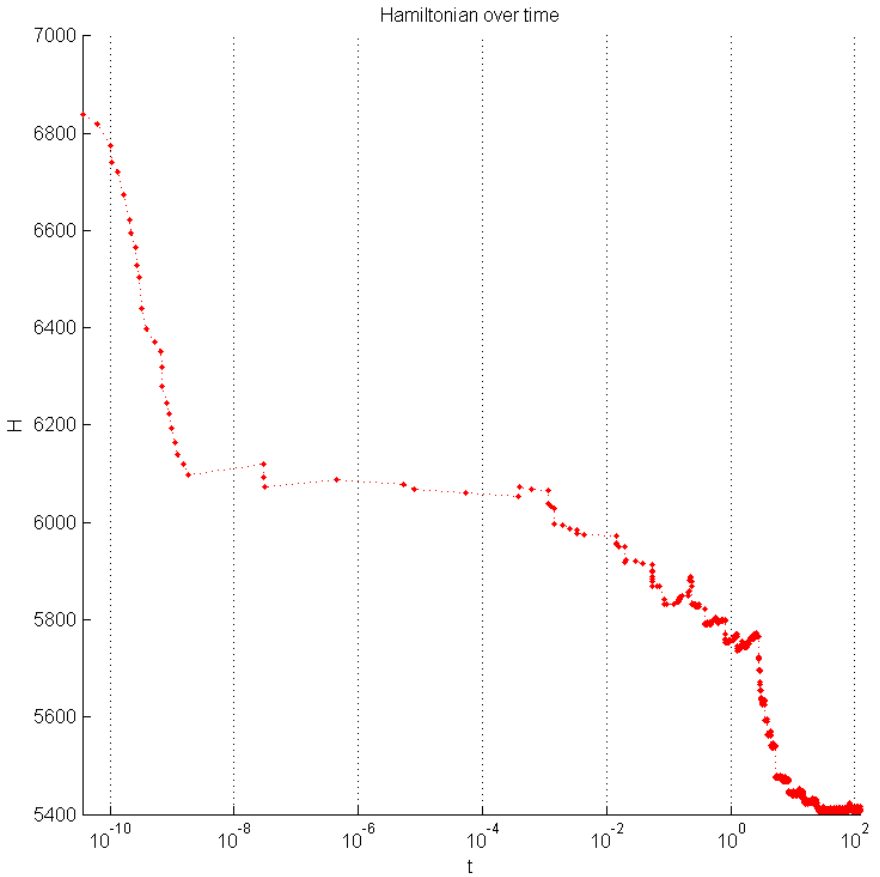

Figure 52: Plot of the Hamiltonian over time for 2000 copies with $\lambda = 1$ and $T = 10$. The t-axis has a logarithmic scale.

5 Discussion

In this paper, a kinetic Monte Carlo implementation of the Cellular Potts Model has been proposed as an alternative of the standard Metropolis algorithm. The main advantage of this new algorithm is the ability to add a time scale to the simulations [11]. Also, in theory, the kinetic Monte Carlo method is more efficient, because many possible copies are processed and thrown away.
The final implementation has the crucial features of the Cellular Potts Model. From the experiments, we can see that the examples converges to the states that are logical and desired. In the end, the relatively easy examples have the configurations with the minimal Hamiltonian and are stable. Also, the higher temperatures cause the Hamiltonian to fluctuate more and be generally higher, along with the fact that the cells are less round in their final configurations. However, the algorithm still has a number of flaws. One of the major shortcomings became apparent when the results were processed and the plots were made. As can be seen in all experiments, a lower temperature causes the Hamiltonian to drop earlier. This has to do with the upper limit of the values of a_i as defined in section 3.3. The upper limit is depending the temperature. A higher temperature causes the upper limit of a_i to decrease. This means that the sum a_0 decreases. As a consequence, τ defined in 3.1 increases too. Thus, when the Hamiltonian drops, this will happen at a later point in time. When the upper limit of a_i would be independent of the temperature, there might be other interesting observations to be made.
Another problem is that the boundary is very sticky. This means there is no adhesion between the boundary and all other cells. During a simulation cells may stick to the boundary, causing the Hamiltonian to drop in an unfair way and influencing the experiments in a bad way. A solution for this problem is to add a special static boundary with a high amount of adhesion. This is not hard to implement, but the problem was found in such a late stadium of the research that this has not been rectified. Moreover, it is not much of an issue as long as the cells don't touch the boundary during the experiments, which has only happened in experiment 1.2.

5.1 Further work

The major objective of future work lies obviously in improving the algorithm. The aforementioned issues can be fixed, along with improving the ease of use and the possibilities of the program. This includes general issues like saving configurations, more support for input files and running more experiments simultaneously. Also,

it would be more tangible to have animations of the simulations, like the Tissue Simulation Toolkit used in Bio-Informatics. Other interesting considerations are the extension to 3D and the use of a six-connected neighborhood on an hexagonal grid instead of an eight-connected (Moore) neighborhood on rectangular grid.
On the technical side, it is necessary to improve the speed of the program for long simulations. At this point, the program is able to handle simulations with 2000 copies in a reasonable time. After this, the program is getting too slow. Like in section 3.3, improving should be done by investigating if more optimizations are possible and implementing them. This also might lead to another intensive collaboration with the creators of MonetDB and SciQL. One optimization might be to further reduce the calling of time consuming functions. Also, there might be ways to transfer some C++ code into SciQL queries to obtain speed benefits, like selecting update μ during step 7 of the algorithm (section 3.1). Most importantly, the interplay between MonetDB, the used system and the implemented program should be researched to find reasons of the major slowdown. An obvious reason would be that the database is stored on the hard disk instead of the main memory and connection would cost much time, but it is not very likely that this is the case.
A more theoretical direction of future research is more extensive experimenting with this new algorithm for the CPM by further exploring the influence of the parameters λ and T. The same goes for the adhesion energies. Then, more similarities or differences between this kinetic Monte Carlo implementation and the standard Metropolis implementation or even other implementations might arise. Also, since the CPM simulations now have a time scale, it would be interesting how the implementation would fare in practice, both in reviewing previous research and in future researches. Futhermore, with a time scale we can attach rates and speed at certain processes, like the extension and retraction of pseudopodia, movement of cells, etc. This increases the ability to compare the cellular Potts model with experimental data, like time-lapse movies of real cells, and to tune the speed of simulations according to this data by changing parameters.

References

[1] J. A. Glazier and F. Graner, *Simulation of the differential adhesion driven rearrangement of biological cells*, Physical Review. E 47, 2128-2154, 1993

[2] R. M. H. Merks, E. D. Perryn, A. Shirinifard, J. A. Glazier, *Contact-inhibited chemotaxis in de novo and sprouting blood-vessel growth*, PLoS Comput Biology 4, 2008

[3] S. D. Hester, J. M. Belmonte, J. Scott Gens, S. G. Clendenon, J. A. Glazier, *A multi-cell, multi-scale model of vertebrate segmentation and somite formation*, PLoS Computational Biology 7, 2011

[4] A. Szab and R. M. H. Merks, *Cellular potts modeling of tumor growth, tumor invasion, and tumor evolution*, Frontiers in Oncology 3, 87 1-12, April 2013

[5] I.Beichl, F. Sullivan, *The Metropolis Algorithm*, Computing in Science & Engineering 2-1, 65-69, 2000

[6] E. Flenner, L. Janosi, B. Barz, A. Neagu, G. Forgacs, I. Kosztin, *Kinetic monte carlo and cellular particle dynamics simulations of multicellular systems*, Physical Review E 85

[7] D. T. Gillespie, *Exact stochastic simulation of coupled chemical reactions*, The Journal of Physical Chemistry 81, 2340-2361, 1977

[8] M. Kersten, Y. Zhang, M. Ivanova, N. Nes, *SciQL, a query language for science applications*, In Proceedings of the first International Array Databases Workshop, 1-12, 2011

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, M. L. Kersten, *MonetDB: two decades of research in column-oriented database architectures*, IEEE Data Data Engineering Bulletin 35, 40-45, 2012

[10] Y. Zhang, M. L. Kersten, M. Ivanova, N. Nes, *SciQL, bridging the gap between science and relational DBMS*, In Proceedings of the 15th International Database Engineering & Applications Symposium, 1-10, 2011

[11] K.E. Sickafus, Kurt E., Kotomin, Eugene A., Uberuaga, Blas P., *Radiation effects in solids*, 1-3, 2007

6 Appendix

6.1 Compilation file

```
#!/bin/sh
libtool --mode=compile --tag=CC g++ -c `env PKG_CONFIG_PATH=
    $INSTALL_DIR/lib/pkgconfig pkg-config --cflags monetdb-mapi` Project
    .cc
libtool --mode=link --tag=CC g++ -o Project `env PKG_CONFIG_PATH=
    $INSTALL_DIR/lib/pkgconfig pkg-config --libs monetdb-mapi` Project.o
./Project
```

6.2 Main application

```
#include <iostream>
#include <fstream>
#include <string>
#include <string.h>
#include <cstdlib>
#include <mapi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

using namespace std;

const int arrayMAX = 25; // The width and height of the world
const int cellsMAX = 70; // Maximum number of cells that can be used

class Life {
        private:
                int heigth, width; // Heigth and width of the view
                int upperleftX, upperleftY; // Upperleft indices of the
                    view
        public:
                //Variables for database purposes
                Mapi dbh;
                MapiHdl hdl;
                MapiHdl hdl2;
                MapiHdl hdl3;
                char *cellcopiedx;
                char *cellcopiedy;
                char *celltargetx;
```

```
char *celltargety;
char *energy;
char *ai;

//Strings for storing the queries
char buffer [1000];
char buffer2 [100000];
int total_buf_size;
int len; //Last two variables keep track of the length
    of the query stored

//Arrays for storing fixed values for computing deltaH
double adhesion[cellsMAX][cellsMAX];
int cellsidealsize[cellsMAX];
int cellscurrentsize[cellsMAX];

//Fixed values for computing deltaH
double temperature;
double compressionresistance;

//Variables for computing statistics
int numberofimages;
double timer;
double speed;

//Variables for checking: if updates table is empty,
//inserted update is the first one and checking if
    update table has already been filled
int check;
bool first;
bool updatesfilledfirsttime;

//Variables for intialization and processing one step
int fromtype;
int totype;
int temptype;
int tempx;
int tempy;

double deltaH; double a;
int fromx;
int fromy;
int tox;
int toy;

double sum;
```

```
                double r1;
                double r2;
                double tau;
                double summation;

                //Functions
                Life();
                void fillInUpdates(Mapi);
                void clean();
                void show();
                void input();
                void changeIdealSize();
                void changeAdhesion();
                void showSizes();
                void oneStep();
                void multipleStep();
                void simulateTime();
                void closeTable(Mapi);
                void output(double, Mapi);
                double computeHamiltonian();
};//Life

//////////////////////////Database functions//////////////////////
void die(Mapi dbh, MapiHdl hdl){
        if (hdl != NULL) {
                mapi_explain_query(hdl, stderr);
                do {
                        if (mapi_result_error(hdl) != NULL)
                                mapi_explain_result(hdl, stderr);
                } while (mapi_next_result(hdl) == 1);
                mapi_close_handle(hdl);
                mapi_destroy(dbh);
        } else if (dbh != NULL) {
                mapi_explain(dbh, stderr);
                mapi_destroy(dbh);
        } else
                fprintf(stderr, "command failed\n");
        exit(-1);
}//die

MapiHdl query(Mapi dbh, char *q){
        MapiHdl ret = NULL;
        if ((ret = mapi_query(dbh, q)) == NULL || mapi_error(dbh) !=
           MOK)
                die(dbh, ret);
        return(ret);
```

```
}//query

void update(Mapi dbh, char *q){
        MapiHdl ret = query(dbh, q);
        if (mapi_close_handle(ret) != MOK)
                die(dbh, ret);
}//update
//////////////////////Database functions//////////////////////


Life::Life() { // default constructor
        dbh = mapi_connect("localhost", 50000, "monetdb", "monetdb", "
            sql", "demo");
        hdl = NULL;
        if (mapi_error(dbh))
                die(dbh, hdl);
        update(dbh, const_cast<char *>("CREATE TABLE updates (
            cellcopiedx INT, cellcopiedy INT, celltargetx INT,
            celltargety INT, energy FLOAT, ai FLOAT, cellcopiedtype INT,
             celltargettype INT);"));
        sprintf(buffer, "CREATE ARRAY theworld (x INT DIMENSION[%d], y
            INT DIMENSION[%d], v INT);", arrayMAX, arrayMAX);
        update(dbh, buffer);

        temperature = 1;
        compressionresistance = 1;

        numberofimages = 0;
        timer = 0.0;
        speed = 0.01;

        updatesfilledfirsttime = false;

        heigth = arrayMAX;
        width = arrayMAX;
        upperleftX = 1;
        upperleftY = 1;
        int i,j;
        for (i = 0; i < cellsMAX; i++){
                cellsidealsize[i] = 15;
                cellscurrentsize[i] = 0;
                for (j = 0; j < cellsMAX; j++){ //Values change upon
                    experimenting
                        if ((i == 0 && j > 40) || (i > 40 && j == 0))
                            adhesion[i][j] = 1;
```

```
                        else if ((i == 0 && j < 40) || (i < 40 && j ==
                            0)) adhesion[i][j] = 1;
                        else if (i < 40 && j < 40) adhesion[i][j] = 1;
                        else if (i < 40 && j > 40) adhesion[i][j] = 1;
                        else if (i > 40 && j > 40) adhesion[i][j] = 1;
                        else adhesion[i][j] = 1;
                }//for
        }//for
        cellscurrentsize[0] = arrayMAX*arrayMAX;

        srand(time(NULL));
}//Life::Life

//Clear the grid
void Life::clean(){
        timer = 0;
        numberofimages = 0;

        update(dbh, const_cast<char *>("UPDATE theworld SET v = 0;"));
        update(dbh, const_cast<char *>("DELETE FROM updates;"));
        cellscurrentsize[0] = arrayMAX*arrayMAX;
        for (int i = 1; i < cellsMAX; i++){
                cellscurrentsize[i] = 0;
        }//for
}//Life::clean

//Print the grid on the screen
void Life::show(){
        cout << endl << "Time: " << timer + (double)numberofimages *
            speed << endl;
        int i,j;
        for(i = upperleftY-1; i < upperleftY+heigth-1; i++) {
                for(j = upperleftX-1; j < upperleftX+width-1; j++) {
                        sprintf(buffer, "SELECT * FROM theworld[%d][%d
                            ];", i, j);
                        hdl = query(dbh, buffer);
                        mapi_fetch_row(hdl);
                        if (atoi(mapi_fetch_field(hdl,2)) == 0) cout <<
                             " -";
                        else if (atoi(mapi_fetch_field(hdl,2)) > 9)
                            printf("%s", mapi_fetch_field(hdl, 2));
                        else printf(" %s", mapi_fetch_field(hdl, 2));
                }//for
                cout << '\n';
        }//for
}//Life::show
```

```
//Read in an external txt file
void Life::input() {
        timer = 0;
        numberofimages = 0;

        cout << "Which number: input....txt?" << endl;
        int number;
        cin >> number;
        sprintf(buffer, "input%d.txt", number);
        ifstream input;
        input.open (buffer,ios::in);
        if ( input.fail ( ) ) {
                cout << "Bestand input.txt ontbreekt! Optie niet
                    beschikbaar." << endl;
        }//if
        else {
                char kar = input.get();
                int i = upperleftY-1;
                int j = upperleftX-1;
                while ( ! input.eof ( ) ) {
                        if (kar == '\n') {
                                i++;
                                j = upperleftX-1;
                        }//if
                        else {
                                if (kar != ' '){
                                        sprintf(buffer, "UPDATE
                                            theworld SET v = %d WHERE x
                                            = %d AND y = %d;", kar - 48,
                                             i, j);
                                        update(dbh, buffer);
                                        if (kar - 48 != 0){
                                                cellscurrentsize[kar -
                                                    48] += 1;
                                                cellscurrentsize[0] -=
                                                    1;
                                        }//if
                                }//if
                                j++;
                        }//else
                        kar = input.get();
                }//while
        }//else
}//Life::input
```

```
//Read the option given in the menu
char readoption() {
        char option;
        option = cin.get();
        while(option == '\n'){
                option = cin.get();
        }//while
        return option;
}//readoption

//Fill in the table updates for the first time
void Life::fillInUpdates(Mapi dbh){
        //Remove all residual updates
        update(dbh, const_cast<char *>("DELETE FROM updates"));

        //Loop over all elements of the array and get the coordinates
           and the type
        sprintf(buffer, "SELECT * FROM theworld WHERE v IS NOT NULL");
        hdl3 = query(dbh, buffer);
        while(mapi_fetch_row(hdl3)){
                fromx = atoi(mapi_fetch_field(hdl3, 0));
                fromy = atoi(mapi_fetch_field(hdl3, 1));
                fromtype = atoi(mapi_fetch_field(hdl3, 2));

                //Select all neighbors of this cell of a different type
                   . Loop over these cells and get the coordinates and
                   the type
                sprintf(buffer, "SELECT * FROM theworld[%d:%d+1][%d:%d
                   +1] WHERE v <> %d;", fromx-1, fromx+1, fromy-1,
                   fromy+1, fromtype);
                hdl2 = query(dbh, buffer);
                while (mapi_fetch_row(hdl2)){
                        tox = atoi(mapi_fetch_field(hdl2,0));
                        toy = atoi(mapi_fetch_field(hdl2,1));
                        totype = atoi(mapi_fetch_field(hdl2,2));

                        //Computer deltaH by looping over all neighbors
                             of this cell of another type and
                           calculating the sum of adhesion energies
                        deltaH = 0;
                        sprintf(buffer, "SELECT * FROM theworld[%d:%d
                           +1][%d:%d+1] WHERE (x <> %d OR y <> %d);",
                           tox-1, tox+1, toy-1, toy+1, tox, toy);
                        hdl = query(dbh, buffer);
                        while (mapi_fetch_row(hdl)) {
```

```
                                temptype = atoi(mapi_fetch_field(hdl,2)
                                    );
                                deltaH += (adhesion[fromtype][temptype
                                    ]*(temptype != fromtype) - adhesion[
                                    totype][temptype]*(temptype !=
                                    totype));
                        }//while

                        //Compute the share of energy in deltaH
                            corresponding to change of cell sizes
                        if (totype == 0) deltaH +=
                            compressionresistance * (2*cellscurrentsize[
                            fromtype] - 2*cellsidealsize[fromtype] + 1);
                        else if (fromtype == 0) deltaH +=
                            compressionresistance * (-2*cellscurrentsize
                            [totype] + 2*cellsidealsize[totype] + 1);
                        else deltaH += compressionresistance * (2*
                            cellscurrentsize[fromtype] - 2*
                            cellscurrentsize[totype] - 2*cellsidealsize[
                            fromtype] + 2*cellsidealsize[totype] + 2);

                        //Compute the corresponding a_i
                        if (deltaH > -20) a = exp( -1*deltaH/
                            temperature);
                        else a = exp( 20/temperature);

                        //Insert the values into the database
                        sprintf(buffer, "INSERT INTO updates VALUES (%d
                            , %d, %d, %d, %f, %f, %d, %d);", fromx,
                            fromy, tox, toy, deltaH, a, fromtype, totype
                            );
                        update(dbh, buffer);
                    }//while
                }//while
                cout << "Initialized\n";
}//Life::fillIn

//Change the target size for a cell
void Life::changeIdealSize(){
        cout << "Cell \tIdeal size" << endl;
        for (int i = 0; i < cellsMAX; i++) cout << i << "\t" <<
            cellsidealsize[i] << endl;
        cout << "Of which cell do you want to change the ideal size?"
            << endl;
        int cell;
        cin >> cell;
```

```
        while (cell >= cellsMAX){
                cout << "This cell does not exist!" << endl;
                cin >> cell;
        }//while
        cout << "What should be the ideal size?" << endl;
        int size;
        cin >> size;
        cellsidealsize[cell] = size;
        cout << "The ideal size of cell " << cell << " is now " << size
            << "." << endl;
}//Life::changeIdealSize

//Change the adhesion energy between two cells
void Life::changeAdhesion(){
        cout << "Adhesionmatrix:" << endl;
        for (int k = 0; k < cellsMAX; k++) cout << "\t" << k;
        cout << endl;
        for (int i = 0; i < cellsMAX; i++){
                cout << i;
                for (int j = 0; j < cellsMAX; j++){
                        cout << "\t" << adhesion[i][j];
                }//for
                cout << endl;
        }//for
        cout << "Between which cells should the adhesion be changed?"
           << endl << "First cell: ";
        int cell1; int cell2;
        cin >> cell1;
        while (cell1 >= cellsMAX){
                cout << endl << "This cell does not exist!";
                cin >> cell1;
        }//while
        cout << endl << "Second cell: ";
        cin >> cell2;
        while (cell2 >= cellsMAX){
                cout << endl << "This cell does not exist!";
                cin >> cell2;
        }//while
        cout << endl << "What should de adhesion be?" << endl;
        double temp; cin >> temp; adhesion[cell1][cell2] = temp;
            adhesion[cell2][cell1] = temp;
        cout << "The adhesion between cell " << cell1 << " and " <<
            cell2 << " is now " << adhesion[cell1][cell2] << "." << endl
            ;
}//changeAdhesion
```

```
//Show sizes of the cells
void Life::showSizes(){
        cout << "Cell \tCurrent size" << endl;
        for (int i = 0; i < cellsMAX; i++) cout << i << "\t" <<
            cellscurrentsize[i] << endl;
}//Life::showSizes

//Simulate one update
void Life::oneStep(){
        //Check if there are any updates - this is the case when the
            grid is one big cell or empty
        hdl = query(dbh, const_cast<char *>("SELECT COUNT(*) FROM
            updates;"));
        mapi_fetch_row(hdl);
        check = atof(mapi_fetch_field(hdl,0));
        if (check == 0) return;

        //Compute the sum of the a_i, two random numbers, tau and
            determine update a_mu
        hdl = query(dbh, const_cast<char *>("SELECT SUM(ai) FROM
            updates;"));
        mapi_fetch_row(hdl);
        sum = atof(mapi_fetch_field(hdl,0));

        r1 = (double)rand()/(double)RAND_MAX;
        r2 = (double)rand()/(double)RAND_MAX;
        tau = (double)1/sum * log((double)1/r1);
        summation = 0.0;
        hdl = query(dbh, const_cast<char *>("SELECT * FROM updates;"));
        while (mapi_fetch_row(hdl)){
                summation += atof(mapi_fetch_field(hdl,5));
                if (r2*sum < summation) break;
        }//while

        //Get the information of the chosen update
        int updatefromx = atoi(mapi_fetch_field(hdl,0));
        int updatefromy = atoi(mapi_fetch_field(hdl,1));
        int updatetox = atoi(mapi_fetch_field(hdl,2));
        int updatetoy = atoi(mapi_fetch_field(hdl,3));
        int type = atoi(mapi_fetch_field(hdl,6));
        int type2 = atoi(mapi_fetch_field(hdl,7));

        //Update the array and the cell sizes
        sprintf(buffer, "UPDATE theworld SET v = %d WHERE x = %d AND y
            = %d;", type, updatetox, updatetoy);
        update(dbh, buffer);
```

```
cellscurrentsize[type]++;
cellscurrentsize[type2]--;

//Delete all update that are not relevant any more after this
    update
sprintf(buffer, "DELETE FROM updates WHERE celltargetx BETWEEN
    %d AND %d AND celltargety BETWEEN %d AND %d;", updatetox-1,
    updatetox+1, updatetoy-1, updatetoy+1);
update(dbh, buffer);

//Create indices for the new updates
int i, j, i2, j2;

//Prepare the string that will store the new updates into the
    table
first = true;
total_buf_size = sprintf(buffer2, "INSERT INTO updates VALUES")
    ;

//Select all elements that are within a range of 2 of the
    changed element, loop over them and get their coordinates
    and their type
sprintf(buffer, "SELECT * FROM theworld[%d:%d+1][%d:%d+1];",
    updatetox-2, updatetox+2, updatetoy-2, updatetoy+2);
hdl2 = query(dbh, buffer);
while (mapi_fetch_row(hdl2)){
        i = atoi(mapi_fetch_field(hdl2,0));
        j = atoi(mapi_fetch_field(hdl2,1));
        fromtype = atoi(mapi_fetch_field(hdl2,2));

        //Select all neigbors of this cells within the range of
            1 of the changed element (so the target of the new
           updates have to be the updated element of a neighbor
            of him),
        //loop over them and get their coordinates and their
           type
        sprintf(buffer, "SELECT * FROM theworld[%d:%d+1][%d:%d
           +1] WHERE x BETWEEN %d AND %d AND y BETWEEN %d AND %
           d AND v <> %d;", i-1, i+1, j-1, j+1, updatetox-1,
           updatetox+1, updatetoy-1, updatetoy+1, fromtype);
        hdl3 = query(dbh, buffer);

        while(mapi_fetch_row(hdl3)){
                i2 = atoi(mapi_fetch_field(hdl3,0));
                j2 = atoi(mapi_fetch_field(hdl3,1));
                totype = atoi(mapi_fetch_field(hdl3,2));
```

```
//Computer deltaH by looping over all neighbors
     of this cell of another type and
    calculating the sum of adhesion energies
deltaH = 0;
sprintf(buffer, "SELECT * FROM theworld[%d:%d
    +1][%d:%d+1] WHERE (x <> %d OR y <> %d);",
    i2-1, i2+1, j2-1, j2+1, i2, j2);
hdl = query(dbh, buffer);
while (mapi_fetch_row(hdl)) {
        temptype = atoi(mapi_fetch_field(hdl,2)
            );
        deltaH += (adhesion[fromtype][temptype
            ]*(temptype != fromtype) - adhesion[
            totype][temptype]*(temptype !=
            totype) );
}//while

//Compute the share of energy in deltaH
    corresponding to change of cell sizes
if (totype == 0) deltaH +=
    compressionresistance * (2*cellscurrentsize[
    fromtype] - 2*cellsidealsize[fromtype] + 1);
else if (fromtype == 0) deltaH +=
    compressionresistance * (-2*cellscurrentsize
    [totype] + 2*cellsidealsize[totype] + 1);
else deltaH += compressionresistance * (2*
    cellscurrentsize[fromtype] - 2*
    cellscurrentsize[totype] - 2*cellsidealsize[
    fromtype] + 2*cellsidealsize[totype] + 2);

//Compute the corresponding a_i
if (deltaH > -20) a = exp( -1*deltaH/
    temperature);
else a = exp( 20/temperature);

//Concatenate the strings
if (first) first = false;
else{
        sprintf(buffer2 + total_buf_size, ",");
        total_buf_size++;
}//else
len = sprintf(buffer2+total_buf_size, "(%d, %d,
     %d, %d, %f, %f, %d, %d)", i, j, i2, j2,
    deltaH, a, fromtype, totype);
total_buf_size += len;
```

```
        }//while
}//while

//Finish the string and add the values to the table
sprintf(buffer2+total_buf_size, ";");
total_buf_size++;
if (total_buf_size > 30) update(dbh, buffer2);

//Renew all other updates belonging to cell type or cell type2
    or both (because of the changed sizes of type and type2)
if (type != 0){
sprintf(buffer, "UPDATE updates SET energy = energy + %f, ai =
    CASE WHEN energy + %f > -20 THEN ai * %f ELSE %f END WHERE
    ((celltargetx NOT BETWEEN %d AND %d) OR (celltargety NOT
    BETWEEN %d AND %d)) AND cellcopiedtype = %d;",
    compressionresistance * 2, compressionresistance * 2, exp
    (-1*(compressionresistance * 2)/temperature), exp(20/
    temperature), updatetox-1, updatetox+1, updatetoy-1,
    updatetoy+1, type);
update(dbh, buffer);
sprintf(buffer, "UPDATE updates SET energy = energy + %f, ai =
    CASE WHEN energy + %f > -20 THEN ai * %f ELSE %f END WHERE
    ((celltargetx NOT BETWEEN %d AND %d) OR (celltargety NOT
    BETWEEN %d AND %d)) AND celltargettype = %d;", -1 *
    compressionresistance * 2, -1 * compressionresistance * 2,
    exp((compressionresistance * 2)/temperature), exp(20/
    temperature), updatetox-1, updatetox+1, updatetoy-1,
    updatetoy+1, type);
update(dbh, buffer);
}//if

if (type2 != 0){
sprintf(buffer, "UPDATE updates SET energy = energy + %f, ai =
    CASE WHEN energy + %f > -20 THEN ai * %f ELSE %f END WHERE
    ((celltargetx NOT BETWEEN %d AND %d) OR (celltargety NOT
    BETWEEN %d AND %d)) AND cellcopiedtype = %d;", -1 *
    compressionresistance * 2, -1 * compressionresistance * 2,
    exp((compressionresistance * 2)/temperature), exp(20/
    temperature), updatetox-1, updatetox+1, updatetoy-1,
    updatetoy+1, type2);
update(dbh, buffer);
sprintf(buffer, "UPDATE updates SET energy = energy + %f, ai =
    CASE WHEN energy + %f > -20 THEN ai * %f ELSE %f END WHERE
    ((celltargetx NOT BETWEEN %d AND %d) OR (celltargety NOT
    BETWEEN %d AND %d)) AND celltargettype = %d;",
    compressionresistance * 2, compressionresistance * 2, exp
```

```
            (-1*(compressionresistance * 2)/temperature), exp(20/
            temperature), updatetox-1, updatetox+1, updatetoy-1,
            updatetoy+1, type2);
        update(dbh, buffer);
        }//if

        //Compute statistics
        timer = timer + tau;
        while (timer > speed){
                //create image using output(dbh, timer);
                numberofimages++;
                timer = timer - speed;
        }//while

}//Life::oneStep

double Life::computeHamiltonian(){
        double hamiltonian = 0;
        sprintf(buffer, "SELECT * FROM updates;");
        hdl = query(dbh, buffer);
        while(mapi_fetch_row(hdl)){
                int type = atoi(mapi_fetch_field(hdl,6));
                int type2 = atoi(mapi_fetch_field(hdl,7));
                hamiltonian += adhesion[type][type2];
        }//while
        hamiltonian = hamiltonian / 2;
        cout << hamiltonian << " en ";
        for(int i = 1; i < cellsMAX; i++){
                if (cellscurrentsize[i] > 0){
                        hamiltonian += compressionresistance * (
                            cellscurrentsize[i] - cellsidealsize[i]) * (
                            cellscurrentsize[i] - cellsidealsize[i]);
                        cout << hamiltonian << endl;
                }//if
        }//for
        return hamiltonian;
}//computeHamiltonian

//Simulate multiple updates
void Life::multipleStep(){
        cout << "How many iterations?" << endl;
        int iterations;
        cin >> iterations;
        sprintf(buffer, "Results/Ham%f-%f.txt", compressionresistance,
            temperature);
        ofstream output (buffer, ios::out);
```

```
        sprintf(buffer2, "Results/Time%f-%f.txt", compressionresistance
            , temperature);
        ofstream output2 (buffer2, ios::out);
        output << computeHamiltonian() << " ";
        output2 << "0 ";
        for (int i = 0; i < iterations; i++){
                oneStep(); cout << i+1 << " : ";
                output << computeHamiltonian() << " ";
                output2 << timer + (double)numberofimages * speed << "
                    ";
        }//for
}//Life::multipleStep

//Simulate for a given time
void Life::simulateTime(){
        cout << "Simulate for how much time?" << endl;
        double time;
        cin >> time;
        sprintf(buffer, "Results/Ham%f-%f.txt", compressionresistance,
            temperature);
        ofstream output (buffer, ios::out);
        sprintf(buffer2, "Results/Time%f-%f.txt", compressionresistance
            , temperature);
        ofstream output2 (buffer2, ios::out);
        output << computeHamiltonian() << " ";
        output2 << "0 ";
        while ((timer + (double)numberofimages * speed) < time){
                oneStep();
                cout << timer + (double)numberofimages * speed << endl;
                output << computeHamiltonian() << " ";
                output2 << timer + (double)numberofimages * speed << "
                    ";
        }//while
}//Life::simulateTime

//Close and remove the tables
void Life::closeTable(Mapi dbh){
        hdl = query(dbh, const_cast<char *>("SELECT * FROM updates;"));
        update(dbh, const_cast<char *>("DROP TABLE updates;"));
        mapi_close_handle(hdl);
        hdl = query(dbh, const_cast<char *>("SELECT * FROM theworld;"))
            ;
        update(dbh, const_cast<char *>("DROP TABLE theworld;"));
        mapi_close_handle(hdl);

        mapi_destroy(dbh);
```

```
}//Life::closeTable

//Generate output for visualisation in Matlab
void Life::output(double time, Mapi dbh){
        sprintf(buffer, "Results/Stats%f-%f-%f.txt", time,
            compressionresistance, temperature);
        ofstream output (buffer, ios::out);
        hdl = query(dbh, const_cast<char *>("SELECT * FROM theworld;"))
            ;
        while (mapi_fetch_row(hdl)){
                if (atoi(mapi_fetch_field(hdl,2)) != 0) output << (atoi
                    (mapi_fetch_field(hdl,2))*15);
                else output << "255";
                if (atoi(mapi_fetch_field(hdl,1)) == arrayMAX-1) output
                     << "\n";
                else output << " ";
        }//while
        output.close();
}//Life::output

//Execute the main menu
void menu(Life & life){
        cout.precision(20);
        char option; // The character will be stored here
        bool stop = false; // When true, the while loop stops
        life.clean();
        while (stop == false){
                cout << endl << "(Q)uit \nShow the (W)orld \n(C)lear \
                    nIn(p)ut \nSet (T)emperature \nSet compression (r)
                    esistance \n(O)ne step \n(M)ultiple steps \nChange (
                    i)deal cell size \nChange (a)dhesion energies \n(S)
                    how current cell sizes \nSimulate for given time
                    using (X)" << endl;
                option = readoption(); // Character being read
                switch (option) {
                        case 'q': case 'Q': // Option quit
                                stop = true;
                                life.output(5.0, life.dbh);
                                life.closeTable(life.dbh);
                                break;
                        case 'w': case 'W': // Option show the world
                                life.show();
                                break;
                        case 'c': case 'C': // Option clean
                                life.clean();
                                life.show();
```

```
            life.updatesfilledfirsttime = false;
            break;
        case 'p': case 'P': // Option input
            life.clean();
            life.input();
            life.show();
            life.updatesfilledfirsttime = false;
            break;
        case 'o': case 'O': // Option one step
            if (!life.updatesfilledfirsttime) life.
                fillInUpdates(life.dbh);
            life.updatesfilledfirsttime = true;
            life.oneStep();
            break;
        case 'm': case 'M': // Option multiple steps
            if (!life.updatesfilledfirsttime) life.
                fillInUpdates(life.dbh);
            life.updatesfilledfirsttime = true;
            life.multipleStep();
            break;
        case 'x': case 'X': // Option simulate for time
            if (!life.updatesfilledfirsttime) life.
                fillInUpdates(life.dbh);
            life.updatesfilledfirsttime = true;
            life.simulateTime();
            break;
        case 't': case 'T': // Option set temperature
            cout << "The current temperature is "
                << life.temperature << ". Change to
                ...? \n";
            cin >> life.temperature;
            cout << "The new temperature is " <<
                life.temperature << ".\n";
            break;
        case 'r': case 'R': // Option set compression
            resistance
            cout << "The current compression
                resistance is " << life.
                compressionresistance << ". Change
                to ...? \n";
            cin >> life.compressionresistance;
            cout << "The new compression resistance
                 is " << life.compressionresistance
                << ".\n";
            break;
```

```
                        case 'i': case 'I': // Option change ideal cell
                            sizes
                                life.changeIdealSize();
                                break;
                        case 'a': case 'A': // Option change adhesion
                            energies
                                life.changeAdhesion();
                                break;
                        case 's': case 'S': // Option show cell sizes
                                life.showSizes();
                                break;
                        default: // Option not known
                                cout << "This is wrong input." << endl;
                    }//switch
        }//while
}//menu

int main(){
        Life life;
        menu(life);
        return 0;
}//main
```