Mathematical Modelling of a Cytoskeletal System: Optimisation of a Plasmid Partitioning Mechanism Using Artificial Evolution

By

Minghua Yin

Supervised by Dr François Nédélec

A Project Report
in Partial Fulfillment of the Requirements for
the Degree of Master of Natural Sciences
at the University of Cambridge

Word Count: 5671

DECLARATION OF ORIGINALITY

'Mathematical Modelling of a Cytoskeletal System: Optimisation of a Plasmid Partitioning Mechanism Using Artificial Evolution'

I understand the University's definition of plagiarism. I declare that, in accordance with Discipline regulation 6, this dissertation is entirely my own work except where otherwise stated, either in the form of citation of published work, or acknowledgement of the source of any unpublished material.

[illegible]

Minghua Yin, Trinity College
04/05/2022

SUMMARY

Recently, computer simulations have been found to be very useful in cell research, with many different modelling tools now in development that aim to shed light on the mechanisms that allow cells to perform complex tasks. Cell division in particular is of great interest as an essential process that sustains life, but the large number of steps and cellular components involved means that it is difficult to gain a detailed understanding of how it works through experiments alone, highlighting the need for computational modelling. This project investigates how a particular artificial cell, modelled using the cytoskeletal simulation tool Cytosim, behaves after its DNA is replicated and the two copies need to be separated so that the cell can successfully divide. The cell was configured to have a relatively simple cytoskeleton consisting of microtubules assisted by plus-end directed motor proteins of a single type. A small number of cell parameters was selected and an optimisation method known as a genetic algorithm was used to look for values for these parameters that would maximise the cell's chances of successful partitioning. The cell's partitioning mechanism was found to be generally effective, often with a success rate of at least 0.9 in the fittest cases, although some possible limitations were identified that could warrant further investigation. The performance of the genetic algorithm was generally good and highly optimal solutions were often found, but certain avenues for further research, regarding both the way the algorithm is coded and the way it is configured, were identified that could help to better explain how the algorithm works and to improve some of the algorithm's less successful aspects.

ACKNOWLEDGEMENTS

I’m extremely grateful to Dr François Nédélec for his constant advice, support, patience and ambition for the whole duration of this project, and to the rest of the Nédélec group for the many thought-provoking chats and meetings.

I’m indebted to Yi Ting for teaching me about reverse-i-search in Bash, without which this project would have been excruciating.

Lastly, I’d like to thank Yuriy Tumarkin and Alejandro Epelde for enthusiastically letting me talk to them about, and show them pictures of, ‘my big green balls’.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GA	Genetic Algorithm
MAP	Microtubule-Associated Protein
MT	Microtubule
MTOC	Microtubule-Organising Centre

1 Introduction

1.1 Computational modelling of cells

Cells are incredibly complex building blocks of life and there remains much to be discovered about their inner workings. In recent decades, computer simulations have been increasingly employed to help better understand the details of cellular processes. For example, successful cell division requires numerous structural proteins, regulatory proteins and enzymes to correctly interact in several steps and checkpoints, and it is difficult to investigate specific mechanisms at high resolution by purely empirical means. Previous work that has benefitted from the use of simulations has included looking at the effect of chromosomal movements and rotations on the acceleration of mitotic spindle assembly (Paul et al., 2009), and showing that dynamic instability, a property of microtubules (see Subsection 1.2), greatly improves the efficiency of locating chromosomes (Holy and Leibler, 1994).

Ultimately, a central aim in computational biology is to continually develop better and more accurate models of cellular processes. Not only is this useful for understanding life, but it can also shed light on *un*natural but successful new mechanisms not found in any known organisms. The potential implications this has in synthetic biology are difficult to predict, but they certainly exist.

This project focuses on cell division, particularly the mechanisms by which genetic material is transported in cells. The main aims of the project were as follows:

- to show that Cytosim, a cytoskeletal simulation tool (see Subsection 4.1 for more details), which uses a number of simplifying assumptions, can reproduce successful mitotic processes with different starting configurations and parameters;
- to improve the simulation tool by coding in simple energy considerations and checkpoints;
- to use a genetic algorithm (see Subsection 4.2 for more details) to optimise sets of parameters for various simulation configurations.

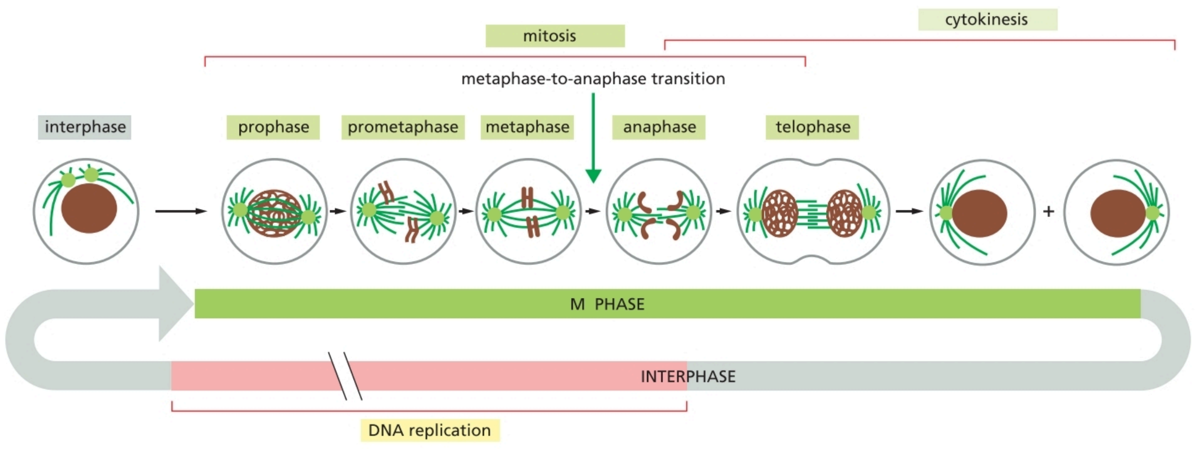


Fig. 1.2.1: The cell cycle for a typical eukaryotic cell (Alberts, 2015, Chapter 17).

1.2 Cell division and the cytoskeleton

Cell division is an essential mechanism by which all organisms reproduce, and thus is key to sustaining life. Although the precise steps vary between different species, all dividing cells must replicate their DNA and then accurately separate the two copies in space before splitting in two. In the eukaryotic cell cycle, this process of DNA partitioning takes place in *M phase* (see Figure 1.2.1), consisting of *mitosis*, during which the DNA is partitioned and nuclear division occurs, and *cytokinesis*, during which the cytoplasm is divided in two and the daughter cells are formed (Morgan, 2007). Similar steps occur in prokaryotic cells such as bacteria. The cell cycle is complex and involves precise activatory and inhibitory processes at various checkpoints by enzymes and regulatory proteins.

The movement of chromosomes and other genetic material in a cell is controlled by the cell's *cytoskeleton*, in particular by filaments such as *microtubules* (MTs) in eukaryotes (Aldaz et al., 2005) and *ParM*, an actin homologue, in bacteria (Salje and Löwe, 2008). MTs are hollow tubes consisting of 13 parallel protofilaments, each of which is made up of 8 nm-long subunits of $\alpha\beta$-*tubulin* (see Figure 1.2.2) (Kerssemakers et al., 2006). The numerous longitudinal and lateral contacts between these tubulin heterodimers make MTs very stiff, and previous experiments have shown that an MT has similar mechanical properties to an elastic rod (Dogterom and Yurke, 1997). MTs are polar, with the *plus-end* polymerising much faster than the *minus-end*. Polymerisation of MTs is greatly assisted by nucleators, typically γ-*tubulin ring complexes*, which act as a blueprint to

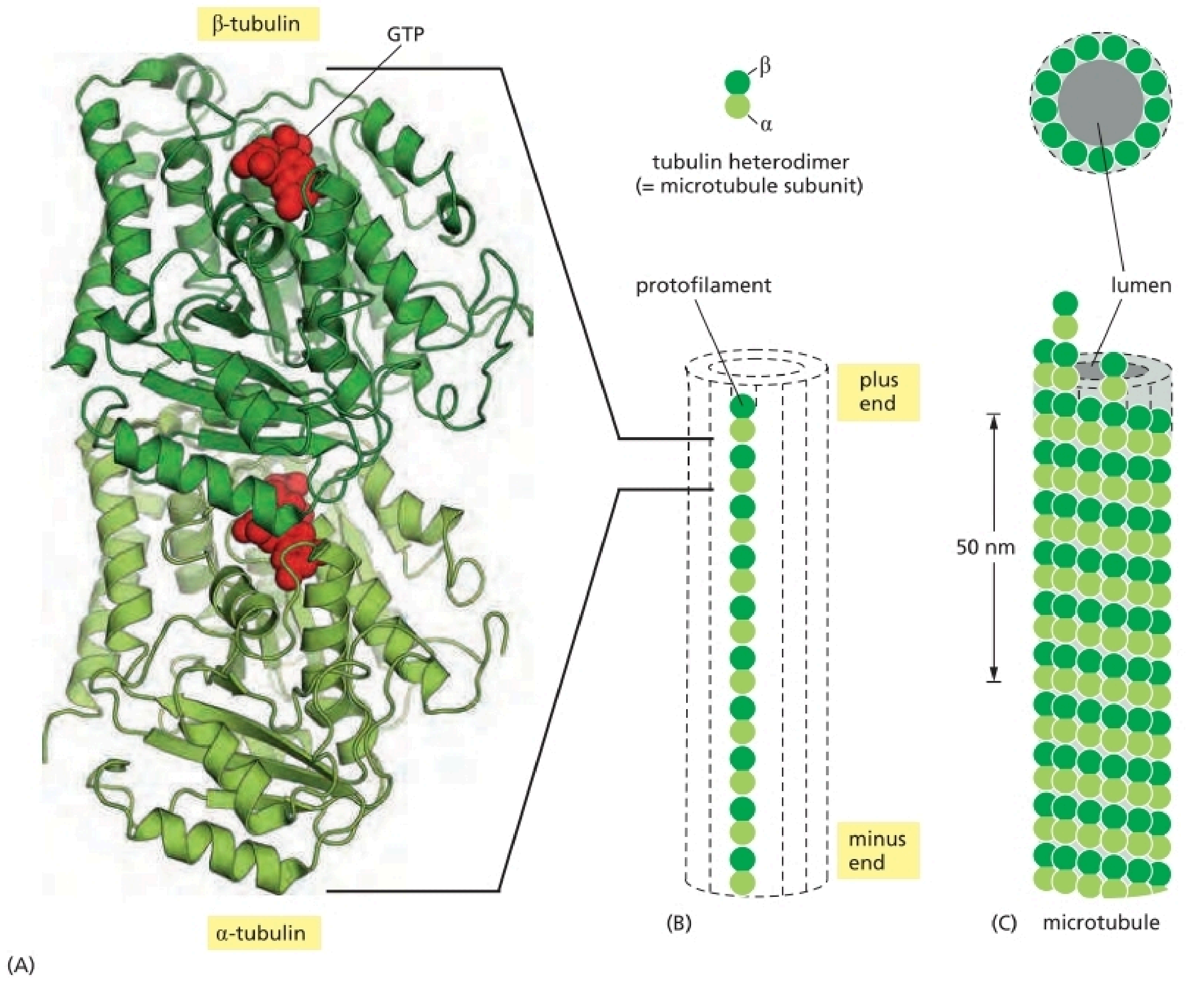


Fig. 1.2.2: The structure of a typical MT (Alberts, 2015, Chapter 16).

which the tubulin heterodimers can attach. MTs are often nucleated from specific locations called *microtubule-organising centres* (MTOCs), resulting in striking, radiating formations called *asters* (De Simone et al., 2016). Animal cells usually have a single MTOC called the *centrosome*, which duplicates before mitosis into two parts that move to opposite sides of the nucleus, forming the poles of the *mitotic spindle*, the structure that separates sister chromatids during mitosis (see Figure 1.2.3). Other eukaryotic cells also have mitotic spindles, although their structures may be somewhat different — for example, the cells of higher plants do not have centrosomes (Stoppin et al., 1994). It is not too difficult to model MTs at the scale of monomers, and models with this resolution are capable of producing reasonably accurate and illuminating results.

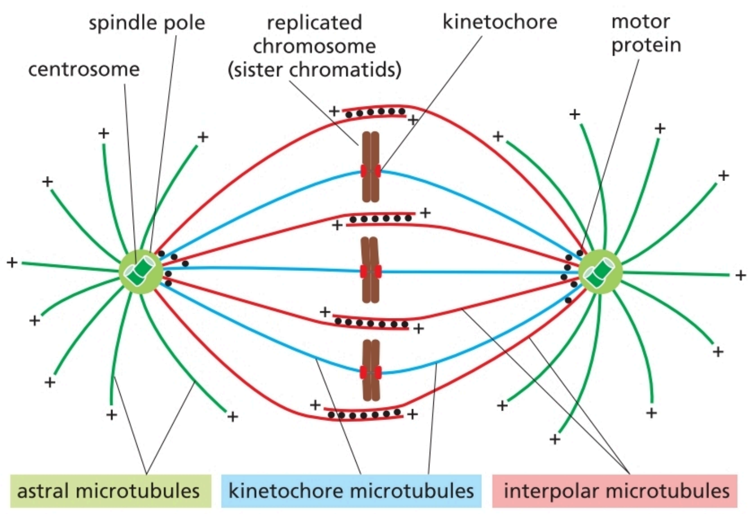


Fig. 1.2.3: The mitotic spindle of a typical animal cell during metaphase, just before the sister chromatids are pulled apart (Alberts, 2015, Chapter 17).

MTs and other filaments exhibit *dynamic instability*, where they can rapidly switch between shrinkage and growth (Hotani and Horio, 1988). The β-tubulin within each subunit is associated with a GTP molecule which is hydrolysed to GDP soon after the subunit attaches to the MT. (In actin and related filaments, ATP is found instead of GTP (Pernier et al., 2016).) Hydrolysis of the GTP reduces the binding affinity of the subunit for neighbouring ones and hence makes it more likely to dissociate from the polymer. An MT will therefore consist mostly of GDP-tubulin monomers, but at each end it is possible for there to be a GTP cap if the rate of addition of subunits at that end is greater than the rate of GTP hydrolysis. A GTP cap favours growth, while an end containing GDP-tubulin favours depolymerisation (by a factor of about 100 compared to GTP-tubulin). MTs are commonly observed to alternate between a state of slow growth and one of rapid depolymerisation. These changes of state are random, with a probability that depends on the concentration of free GTP-tubulin subunits. A transition from growth to shrinkage is called a *catastrophe* and a transition from shrinkage to growth is called a *rescue*.

Eukaryotic cells contain many different types of *microtubule-associated proteins* (MAPs), which attach to MTs and alter their behaviour, such as by initiating rescue or catastrophe, or by helping to transport the MTs through the cytoplasm (Galjart, 2010). In particular, *kinesins* and *dyneins* are *motor proteins* that bind to and move along MTs. Most (but not all) kinesins move towards the plus-end of an MT while dyneins are minus-end directed (Kikkawa, 2013). Some kinesins can

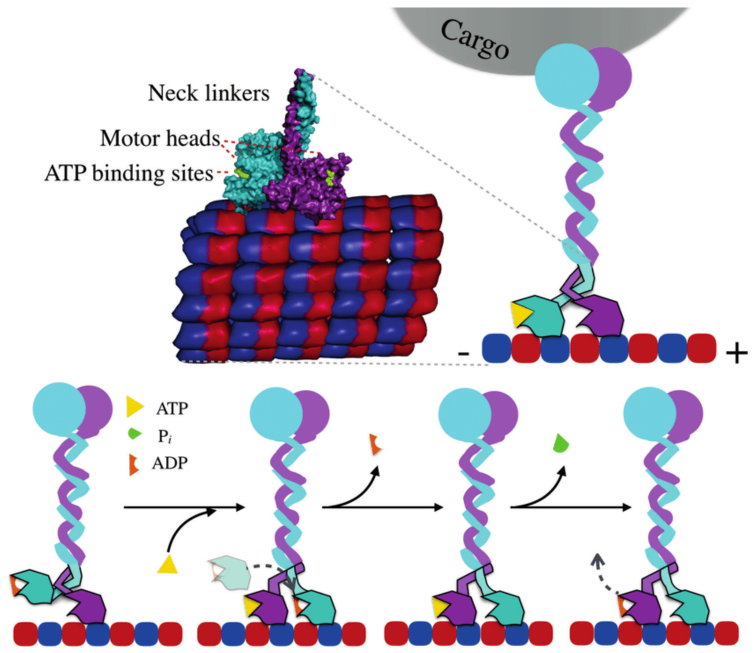


Fig. 1.2.4: The walking mechanism by which kinesin moves along an MT (Korosec and Forde, 2017).

bind to multiple MTs, while others have sites to which intracellular cargo such as chromosomes can bind (Verhey et al., 1998). Kinesins typically have two heavy chains, each of which has a tubulin-binding domain (a 'foot') and a site that binds ATP (Howard et al., 1989). The two feet of a kinesin molecule attach to adjacent heterodimers on an MT. Kinesin moves via a 'walk' (see Figure 1.2.4): hydrolysis of one ATP molecule causes the back foot to detach from the MT, swing forwards and reattach in front of the other foot (Vale and Milligan, 2000). Hence the length of each step that kinesin makes is 8 nm. Motor proteins attach to MTs at a rate that depends on their concentration, while their detachment is affected by the force experienced by the molecular link in the motor protein, e.g. due to tension from cargo. The relationship between the detachment rate k_{off} and the magnitude f of the experienced force is exponential and obeys *Bell's law*, $k_{\text{off}} = k_0 \exp(f/f_0)$, where k_0 is the load-free rate and f_0 is a characteristic force that depends on temperature and a molecular scale (Walcott, 2008). Intracellular events such as motor protein detachment that can be described using well-defined equations can be readily modelled.

Motor proteins are essential for the action of the mitotic spindle. Dynein and different types of kinesin help to organise the spindle and transport cargo along MTs (Verhey et al., 2011). For example, in animal cells, kinesin-5 binds to two antiparallel MTs originating from opposite poles and pushes the MTs part, since each pair of feet moves towards the plus-end of its respective MT. Kinesin-4 and kinesin-10 push attached chromosomes towards the plus-ends of MTs. Kinesin-14 is minus-end directed and tends to pull the poles together, and dynein links MTs with the actin cytoskeleton at the cell cortex, providing anchorage to the poles (Walczak et al., 2010). Despite each motor protein having a simple action, together they allow the spindle to perform important and complex tasks such as organising chromosomes and separating sister chromatids. Many other MAPs also play important roles — for example, proteins such as CLASP and CLIP-170 bind to MTs and promote rescue (Fees and Moore, 2019). The simplicity of each motor protein's action also means that it is relatively straightforward to make computational models of the cytoskeleton that can reproduce the complex behaviour observed in real cells. It is worth noting that, although the focus in this section has been on animal cells, the cytoskeletons of other eukaryotic cells and prokaryotic cells will have many similarities to the above, and the basic concepts described here are applicable to cells from a wide variety of species.

2 Results

A significant amount of time in this project was spent working on modifications to the Cytosim code that were required for the simulations (see Subsection 4.1.3) and completing the code for the genetic algorithm (GA) (see Subsection 4.2.2). The utility of the GA was then tested with a large number of simulations. Examples of the configuration files used in the project, for both the simulations and the GA, can be found in Appendix B.

The simulations are of an artificial, rod-shaped cell, similar in size to a typical bacterial cell. Inspiration was loosely taken from such single-celled organisms as the yeast *Schizosaccharomyces pombe* and the bacterium *Escherichia coli*, forming a sort of 'hybrid' cell, with its (already replicated) DNA found in two plasmids (modelled as single-point `beads`), but with MT `fibers` as in eukaryotes. The cell has two types of protein: `complexes`, which are `couples` consisting of plus-end directed `kinesins` (similar to kinesin-5), and `nucleators`, which are `singles`. Instead of the cell having a specialised centrosome, the `nucleators` are directly attached to the plasmids so that MTs are nucleated with their minus-ends bound to the plasmids. `Complexes` are synthesised free-floating in the cytoplasm and can bind to a pair of antiparallel MTs, one from each plasmid. Since the `kinesins` are all plus-end directed, a doubly attached `complex` will push the MTs away from each other and hence move the plasmids further apart — this is the primary mechanism by which the artificial cell aims to separate its DNA. The MTs exhibit dynamic instability with a hydrolysis rate of 1 s^{-1}, which corresponds to a catastrophe rate of approximately 0.12 s^{-1} (see Subsection 4.1.2).

Table 2.0.1 gives the exact specifications of the cell and its components. Most of the values were taken from empirical data from past research (Campbell and Mullins, 2007; Footer et al., 2007; Garner et al., 2004; Gittes et al., 1993).

This hybrid cell was designed to strike a good balance between simplicity, given the time constraints of the project and the limitations of Cytosim itself, and complexity, so that the cell's mechanisms would not be too distant from reality and a significant step could be made towards the long-term aim of modelling and optimising more complex cells (see Subsection 3.2). A rod shape was chosen (rather than, say, a sphere) as it provided a clear, simple way of determining whether

Property	Value	Property	Value
Cell		**Nucleator**	
Radius	0.4 μm	Stiffness	1000 pN μm^{-1}
Length	2 μm	Nucleation rate	1 s^{-1}
Viscosity	0.1 Pa s	Init length of created MT	0.01 μm
Plasmids		**Kinesin**	
Confinement stiffness within cell	100 pN μm^{-1}	Binding rate	10 s^{-1}
MTs		Unbinding rate	0.3 s^{-1}
Rigidity	20 pN μm^{2}	Unbinding force	2.5 pN
Confinement stiffness within cell	100 pN μm^{-1}	Unloaded speed	0.02 μm s^{-1}
Unit length	0.008 μm	Stall force	6 pN
Shrinking speed	0.5 μm s^{-1}	**Kinesin complex**	
Hydrolysis rate	1 s^{-1}	Stiffness	100 pN μm^{-1}
Growing force	1.7 pN	Diffusion coefficient	0.05 μm^{2} s^{-1}
Total # monomer units available	200	Natural length	0.025 μm

Table 2.0.1: The specifications for the artificial, rod-shaped cell used for the simulations. A few modifications were made for some of the simulations — these are explicitly stated in the relevant sections.

Property	**Range**
Growing speed of MTs (μm s^{-1})	$[0.05, 0.5]$
Binding range for `kinesin` (μm)	$[0.01, 0.05]$
Number of `nucleators` per plasmid	$[1, 12]$
Number of `complexes`	$[50, 5000]$

Table 2.1.1: The parameters that the GA varied for the simulations in Subsection 2.1.

a partitioning was successful: the long axis of the cell was positioned to coincide with the x-axis, with the cell centred at the origin, so that the plane $x = 0$ divided the cell evenly. The partitioning could then be considered successful if at the end of the simulation the plasmids were on opposite sides of this plane.

The two DNA `beads` are initially placed along the x-axis, at $x = -0.3$ and $x = -0.5$ respectively (units are μm). The simulations were configured with a time step of 0.001 s and a runtime of 100 s, which is the order of magnitude within which the cell should be able to partition its DNA.

The GA was configured with a population size per generation of 50, and was set to terminate (see Step 3 in Figure 4.2.1) either (a) if the maximum fitness in the most recent generation is greater than 0.9 or (b) after 50 generations.

2.1 Infinite energy case

Initially, four properties were decided on for the GA to optimise. These, along with their specified ranges, are listed in Table 2.1.1. The ranges were chosen to be realistic and consistent with empirical data for naturally occurring cells of a similar size (Verhey et al., 1998; Howard et al., 1989; Holy and Leibler, 1994; Matsuyama et al., 2006).

For these simulations, no modifications were made to the base Cytosim code. The GA was first run with a population size of 50 and with 40 simulations for each set of parameters (aka individual). The fitness function was defined as follows: for each of the 40 simulations, the final x-coordinates of the plasmids are examined. If they have opposite signs, then the partitioning was a success and this particular simulation gets a score of 1. Otherwise, the simulation gets a score of 0. A mean

Property	**Ind 1**	**Ind 2**
Growing speed of MTs (μm s^{-1})	0.2653	0.1824
Binding range for `kinesin` (μm)	0.0473	0.0447
Number of `nucleators` per plasmid	9	11
Number of `complexes`	2360	4379

Table 2.1.2: Two of the perfect scorers for the infinite energy simulation, demonstrating different optimal solutions.

score over all 40 simulations is then calculated, which corresponds to the fitness of the individual.

The GA terminated at just generation 0 and found four individuals with a perfect fitness score of 1, two of which are shown in Table 2.1.2. Further runs of the GA yielded yet more optimal solutions, indicating that the ranges from Table 2.1.1 are capable of producing positive results. Additions to the Cytosim code, and eventually the cell configuration, were then made to model the cell more realistically, as detailed in the next Subsections.

2.2 Additional energy constraints

The Cytosim code was modified to model simple energy constraints. As described in Subsection 4.1.3), the cell was given an energy `income` and the energy costs of three key actions were specified.

- The synthesis of a `complex` was set to cost 400 ATP.
- The movement of a `kinesin` by one step was set to cost 1 ATP. Since each step is 8 nm long, the movement cost per μm is 125 ATP.
- The polymerisation of an MT was set to cost 13 GTP (ATP equivalent) per monomer length (8 nm). The simulation was set up so that all 200 monomer units are available to the cell right from the start, but this cost accounts for the replenishment of one GTP molecule per attached heterodimer (to replace the GTP that is hydrolysed), with each MT consisting of 13 protofilaments. The movement cost per μm is thus 1625 ATP equivalent.

A very rough, order-of-magnitude estimate of an appropriate `income` was calculated by taking one of the optimal solutions from Subsection 2.1 (Ind 1 from Table 2.1.2) and estimating the energy used by the cell during the simulation. With $v_{\text{grow}} = 0.2653$, $n_{\text{nuc}} = 9$ and $n_{\text{com}} = 2360$, and the assumption that each `kinesin` travels a total of 1 μm, the energy use per second E was estimated to be about 260000 units of ATP[1].

The configuration was slightly changed so that the `complexes` would now be added to the cell one by one, at a rate in the range $[10, 100]$ s^{-1}, to roughly match the range for the number of `complexes` in Subsection 2.1. This parameter replaced the previously used number of `complexes`.

The GA was run with these modifications, but it was found that the `income` was too low for the number of successful partitionings to be significant. It is important to emphasise that the estimate obtained above is extremely loose and so it is justified to change the leading digit or increase the energy intake by a reasonable amount. Once the `income` was increased to at least 800000 ATP per second, the results became much more noteworthy. The results below are all from simulations where the `income` was 800000 ATP per second.

Again with a population size of 50 and 40 simulations per individual, the GA terminated at generation 16 with a maximum fitness of approximately 0.925[2]. Table 2.2.1 shows the individual with this fitness. The frames in Figure 2.2.1 show a graphical rendering of one successful simulation of Ind 3. The heatmap in Figure 2.2.2 shows the distribution of fitnesses for each generation, and the plotted lines show the maximum fitnesses and average fitnesses across all generations. The increase in both maximum and average fitness is noticeable, demonstrating correct function of the GA in producing generally fitter individuals from one generation to the next. Sometimes the maximum fitness decreased slightly from one generation to the next, due to no individuals being copied into the next generation via elitism (see Figure 4.2.1).

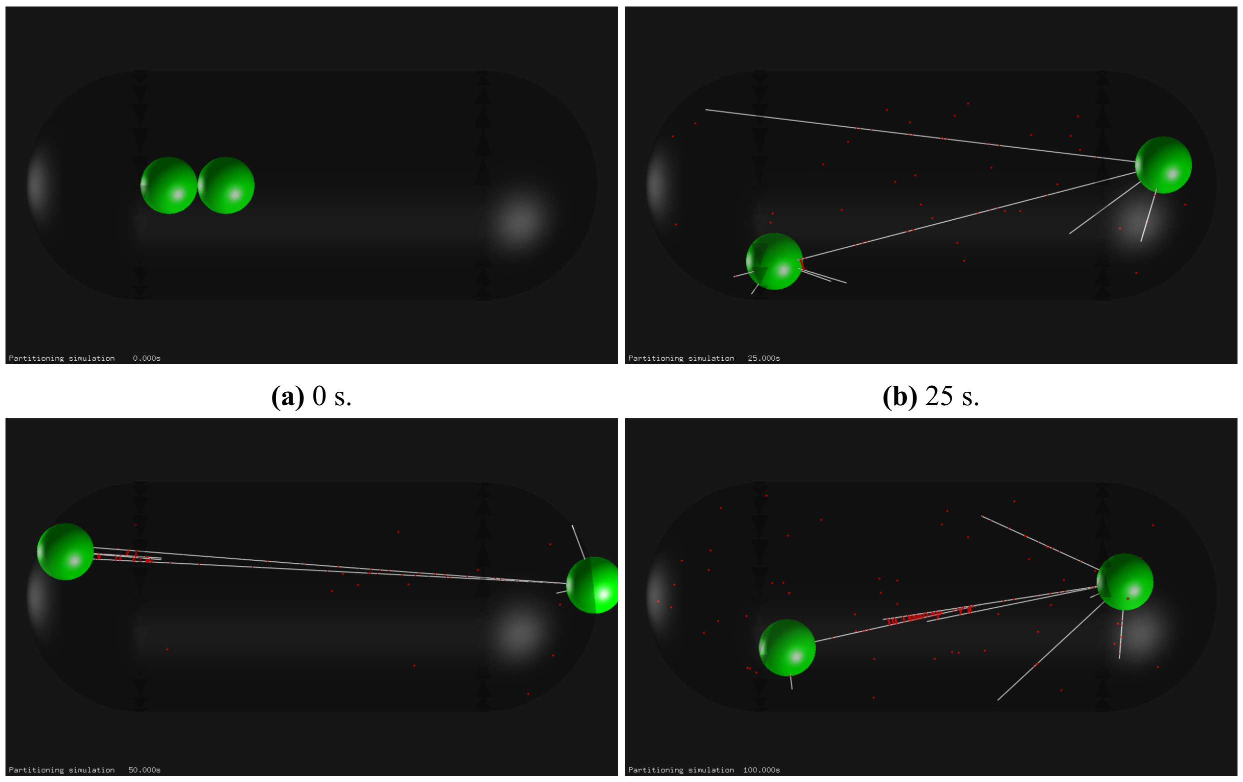
The GA was then rerun, but this time with a population size of 200 and 10 simulations per individual, so that the total number of calculations made by the GA stayed the same as before. This time, an individual (shown in Table 2.2.2) with a perfect fitness of 1 was found in generation 0. Further runs of the GA, with both evolutionary configurations, yielded similar results, with the second configuration consistently producing generations with higher maximum fitnesses earlier on,

[1]The full equation used was $E = 2n_{\text{nuc}} \cdot 400(v_{\text{grow}} + 0.01)/0.008 + n_{\text{com}}(400 + 2/0.008)/100$.

[2]There was a small bug in the Cytosim code which meant that the number of simulations was occasionally slightly less than 40.

Property	**Ind 3**
Growing speed of MTs (μm s^{-1})	0.4665
Binding range for `kinesin` (μm)	0.0500
Number of `nucleators` per plasmid	12
`Complex` synthesis rate (s^{-1})	12.8571

Table 2.2.1: The fittest individual when the GA was run with a population size of 50 and 40 simulations per individual.


Partitioning simulation 0.000s
Partitioning simulation 25.000s
Partitioning simulation 50.000s
Partitioning simulation 100.000s

(a) 0 s. **(b)** 25 s.

(c) 50 s. **(d)** 100 s.

Fig. 2.2.1: Frames of a successful simulation for Ind 3 from Table 2.2.1. The two plasmids are represented by the green balls, which are purely visual as the plasmids are actually modelled as single-point `beads`. The red dots are `complexes`, some of which are free-floating in the cytoplasm and some of which are bound to one or two MTs.

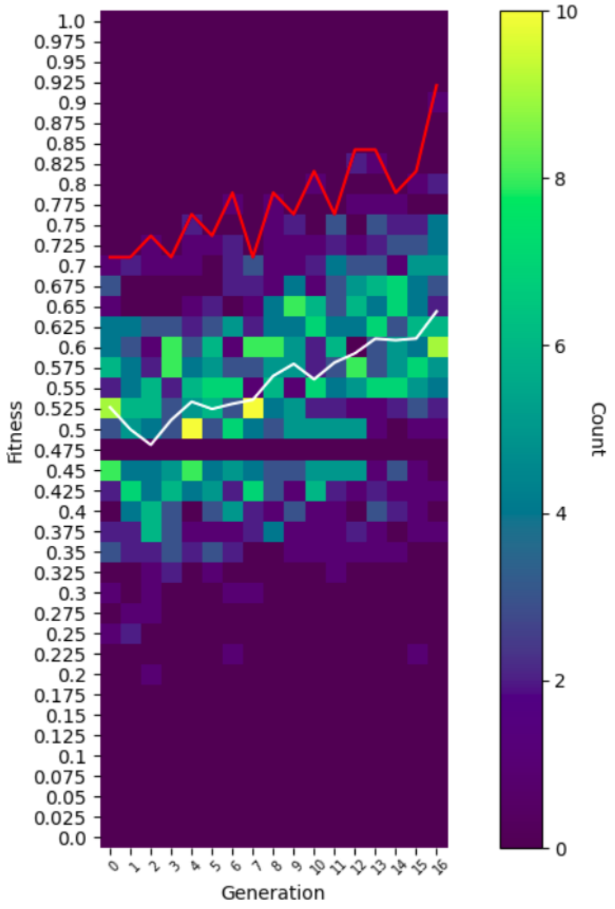


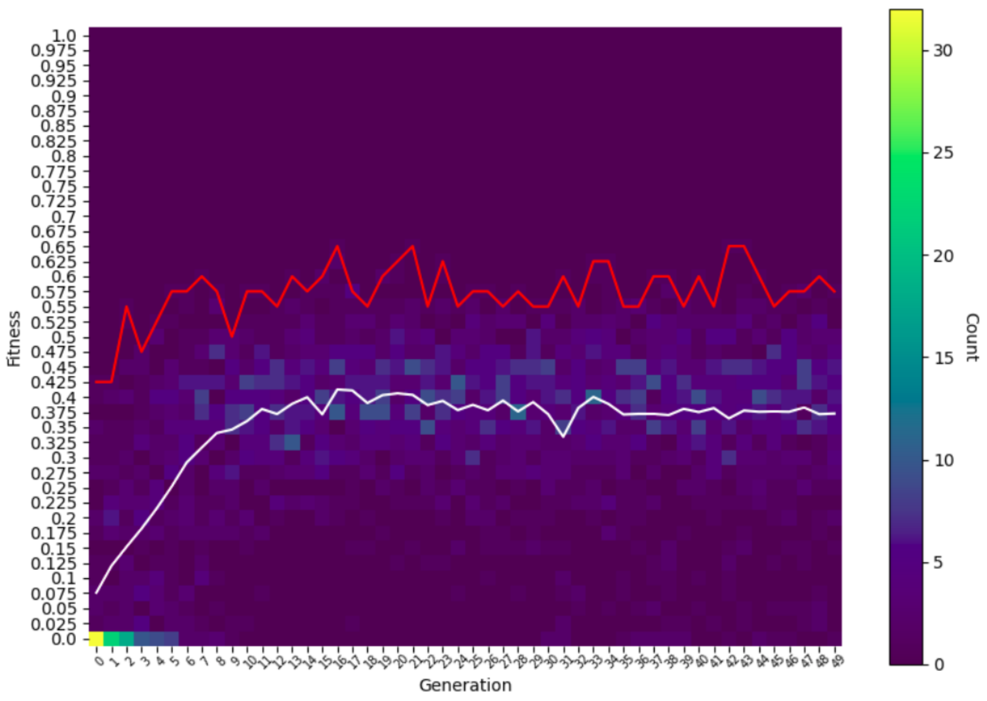
Fig. 2.2.2: Heatmap and fitness plots for runs of the GA with a population size of 50 and 40 simulations per individual. The red line is the maximum fitness plot and the white line is the average (mean) fitness plot.

Property	**Ind 4**
Growing speed of MTs (μm s^{-1})	0.4400
Binding range for `kinesin` (μm)	0.0367
Number of `nucleators` per plasmid	11
`Complex` synthesis rate (s^{-1})	94.2857

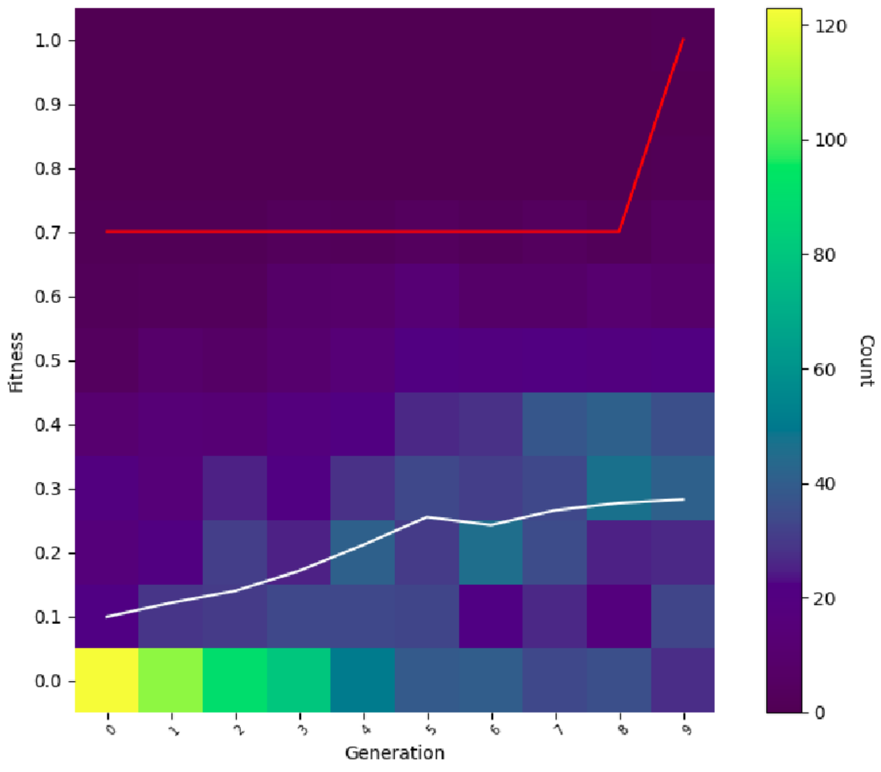
Table 2.2.2: The perfect scorer when the GA was run with a population size of 200 and 10 simulations per individual.

often in generation 0, compared with the first configuration.

To further test both the GA and the suitability of the artificial cell's partitioning mechanism, the viscosity of the cell was increased from 0.1 Pa s to a very high 100 Pa s, rendering the plasmids virtually immobile unless pushed by the cell's MT-`complex` system (the 'spindle'). Figure 2.2.3 shows the heatmaps and plots for typical GA runs with the first and second evolutionary configurations respectively. With the first configuration, the GA typically terminated at generation 49 (the maximum possible generation before the GA is programmed to stop) with a maximum fitness that rarely exceeded 0.625. While the average fitness increased as generation number, the change (if any) in maximum fitness was much less, with perhaps only a slight increase compared to the very first generations. A more successful GA would be expected to produce a much clearer improvement in the maximum fitness over generations. The difference in success between the first and second configurations is very stark, with the second configuration often producing individuals with very high fitnesses after a small number of generations and a more significant increase in maximum fitness over generations. Even given this unrealistic and extreme viscosity, the maximum fitnesses achieved by the GA indicate that the cell's spindle significantly improves the cell's chances of successful partitioning, since without it it would be expected that the fitness of any individual would be close to 0 for these simulations.



(a) Population size of 50, 40 simulations per individual.



(b) Population size of 200, 10 simulations per individual.

Fig. 2.2.3: Heatmaps and fitness plots for runs of the GA with the cell configured to have a very high viscosity of 100 Pa s. In each Subfigure, the red line is the maximum fitness plot and the white line is the average (mean) fitness plot.

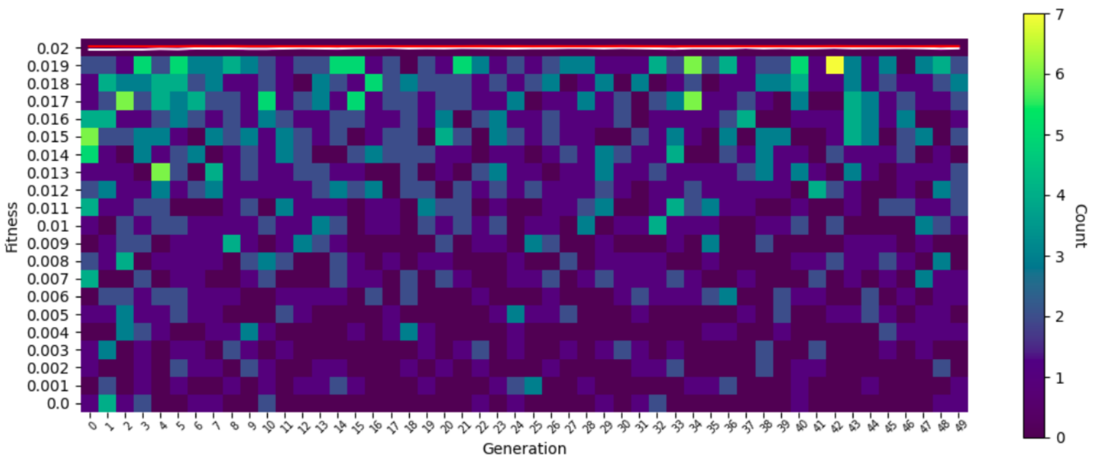
2.3 Partitioning checkpoint

On inspection of the final x-coordinates of the plasmids for many different simulations, it was discovered that sometimes the plasmids were still very close to the origin, and to each other, at the end ($|x| < 0.1$, say). Real cells typically separate their DNA further from each other so that the daughter cells are better defined and successful cytokinesis can occur. The next addition to the Cytosim code was a checkpoint at each time point that checks if (a) the plasmids are on opposite sides of the cell and (b) $|x| > 0.5$ for each plasmid — if so, the cell is considered to have successfully partitioned its DNA. If this still has not happened after 100 s, the cell is deemed to have failed. This was inspired by real checkpoints that are found at key moments in the cell cycle in most species, with the cell cycle unable to pass a checkpoint until the conditions there are met (Hartwell et al., 1974).

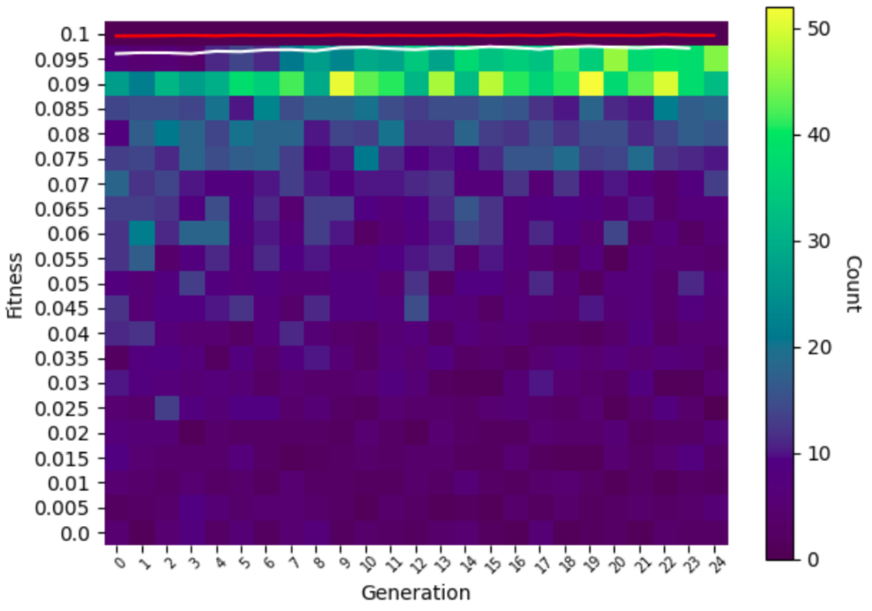
The fitness function for the GA was also modified: now, if partitioning is unsuccessful, the simulation gets a score of 0. If partitioning is successful, the time t taken to partition determines the score s ($0 < s < 1$) via a linear scale, with a smaller t corresponding to a higher score. Explicitly, $s = 1 - t/100$.

The GA was run multiple times with both evolutionary configurations from before. Figure 2.3.1 shows typical results for the two configurations respectively. Regardless of the evolutionary configuration, most of these runs did not produce a significant improvement in maximum fitness over generations, and any improvements in average fitness appeared to be small. Closer inspection of some of the fittest individuals (see Table 2.3.1) showed a very mixed picture and significant inconsistencies: for example, simulations of Ind 5 were sometimes very short, with some successful partitionings occurring in less than 10 s (see Figure 2.3.2), but often the DNA did not successfully separate at all. One interesting feature that was discovered when viewing some of the graphical outputs was that in some successful simulations, especially when the number of `nucleators` was low, no `complexes` ever attached to two MTs and hence there was no pushing action on the MTs from the motor proteins. Despite this, successful partitioning was still sometimes achieved, apparently because the growing MTs were able to push the plasmids towards the ends of the cell.

Similarly to what was found in Subsection 2.2, the second evolutionary configuration again performed much better than the first in finding individuals with notably higher fitnesses more quickly.



(a) Population size of 50, 40 simulations per individual.

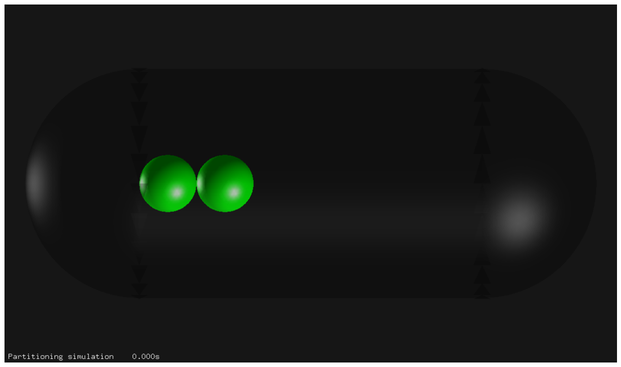


(b) Population size of 200, 10 simulations per individual.

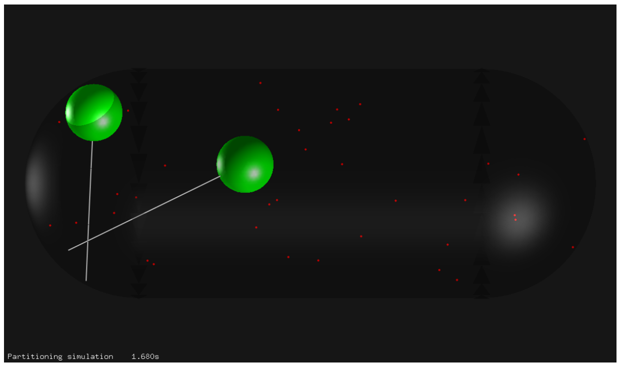
Fig. 2.3.1: Heatmaps and fitness plots for runs of the GA with a partitioning checkpoint added to the cell. In each Subfigure, the red line is the maximum fitness plot and the white line is the average (mean) fitness plot.

Property	**Ind 5**	**Ind 6**
Growing speed of MTs (μm s^{-1})	0.4612	0.4682
Binding range for `kinesin` (μm)	0.0340	0.0180
Number of nucleators per plasmid	1	1
`Complex` synthesis rate (s^{-1})	20.0000	62.8571
Fitness	0.024100	0.097320

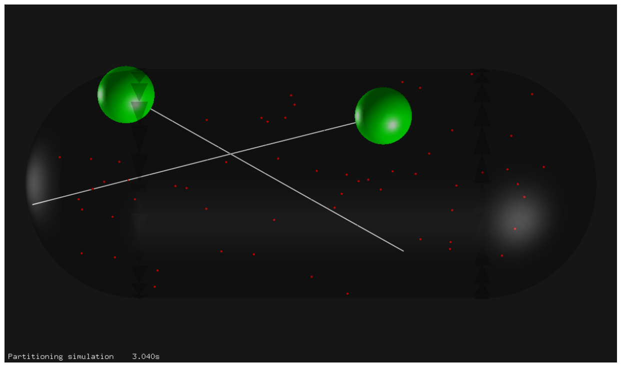
Table 2.3.1: One of the fittest individuals from each of two runs of the GA. Ind 5 is from a run with the first evolutionary configuration, and Ind 6 is from a run with the second evolutionary configuration.



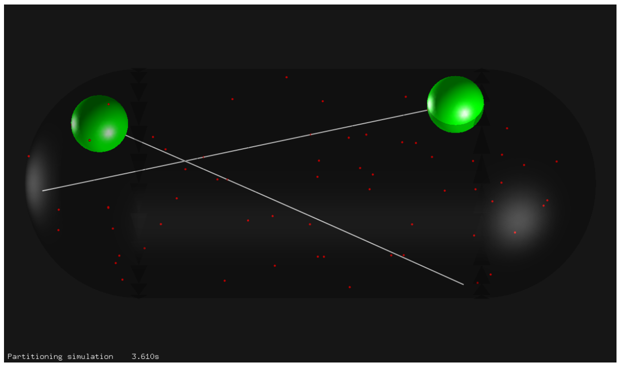
(a) 0 s.



(b) 1.68 s.



(c) 3.04 s.



(d) 3.61 s — checkpoint passed.

Fig. 2.3.2: Frames of a successful simulation for Ind 5 from Table 2.3.1. Interestingly, no `complexes` attached to both MTs! The single MT attached to each plasmid seemed to be enough to push the plasmids towards the ends of the cell, fortunately to opposite ends.

3 Discussion

A large number of simulations, all of the same basic cell but with somewhat different configurations, was successfully run and produced results of varying quality. Given that the use of relatively specialised cellular simulation tools such as Cytosim is currently burgeoning, and that up until now GAs have rarely been used for such problems as DNA partitioning, the results from this project are very promising and naturally inspire further avenues of investigation.

3.1 Conclusions

Even with relatively simple individual mechanisms, the components of a cell exhibit self-organisation that allows the cell to perform important tasks

In the artificial cell, each component has a small number of simple actions: for example, a `kinesin` can bind to an MT, unbind from an MT or walk along an MT; and an MT can grow or shrink. Given the right parameters, these components work together and communicate indirectly with each other to produce well-organised systems. In most of the simulations, it was observed that eventually the MTs attached to each plasmid tended to be longer on the side facing the other plasmid, and tended to align themselves with the long axis (x-axis) of the cell. The shape of the resulting structure consisting of connected plasmids, MTs and `complexes` is very reminiscent of the structures observed in real cells. The simulations confirm that no sophisticated communication or interaction between a cell's components is required for complex larger-scale behaviour to occur.

Computational models are a very useful and efficient way to investigate cellular dynamics

It was demonstrated that computer simulations can produce artificial cells with a strong likeness to natural ones. Making adjustments to a simulation is much easier, faster and cheaper than trying to engineer a real cell, and parallel simulations can be fast-forwarded and run much more speedily than physical experiments. While wet laboratory work is undeniably essential in biological research, computational models can be of great assistance in improving our understanding of various cellular mechanisms.

The artificial cell considered in this project is reasonably successful at partitioning its DNA

In many of the runs in Subsections 2.1 and 2.2, parameters that resulted in (almost) perfect partitioning behaviour were found very early on, even when energy constraints were introduced. For the normal viscosity runs in Subsection 2.2 with the first evolutionary configuration, the typical maximum fitness found was somewhat lower but still above 0.8, which is comparable to (perhaps slightly worse than) success rates typically found in nature. The effectiveness of the cell's spindle was very clearly demonstrated by the results from the high viscosity runs, where despite the unrealistic conditions a maximum fitness of over 0.5 was still attained.

After the partitioning checkpoint was added for the simulations in Subsection 2.3, the performance of the cell was not as consistent. There are a few possible reasons for this. Firstly, it is possible that the cell's mechanism has certain limitations that prevent it being able to consistently push its plasmids far enough apart to pass the checkpoint — indeed, examination of frames of some of these simulations showed that regular unbinding of `complexes` and MT catastrophes did often stall the separation of the plasmids, decreasing the effectiveness of the cell's mechanism. Secondly, the checkpoint may have been too strict, and a greater success rate might be observed if the $|x| > 0.5$ threshold were relaxed a little. Thirdly, the fitness function could certainly be improved, as the linear scale used meant that a high score could only be obtained if the cell managed to partition its DNA within tens of seconds, which is not very realistic. A function with a flatter peak such as $s = 1 - (t/100)^3$ or similar could probably produce better scaled fitnesses. All of these possibilities could be investigated by making further tweaks to the configurations of the cell and the GA.

The effectiveness of a GA is significantly affected by the way it is configured

The stark difference in performance between the two evolutionary configurations in Subsections 2.2 and 2.3 shows that it is important to strike a good balance between population size and number of replicates (or other parameters that affect the total runtime of the GA). Despite the second configuration having a smaller number of simulations per individual, the much larger population size cast out a much wider net and drastically increased the ability of the GA to find more optimal solutions. However, if the number of simulations is too low, then the errors in the calculated fitnesses are high, potentially leading to inaccurate results. It would certainly be interesting to experiment with different evolutionary configurations to see if an even more optimal one can be found for these simulations.

Other parameters in the evolutionary configuration, or certain details of the GA, could also be tweaked, which might lead to further improvements in the GA. For example, it would be interesting to see what would happen if the elitism proportion were increased and the mutation proportion decreased (see Figure 4.2.1 for further explanation), or vice versa. Some GAs perform crossover at multiple sites rather than just one (Mitchell, 1998, Chapter 1) — it could be worth investigating what effect this change would have.

Perhaps one suboptimal feature of the GA used in this project is that, for some of the runs, while the average fitness is often significantly improved over generations, the improvement in maximum fitness tends to be less great. Further investigation would be required to determine to what extent this issue is caused by the artificial cell's partitioning mechanism, and to what extent by the GA. For example, increasing the elitism proportion would, at the very least, reduce the decreases in maximum fitness that sometimes occur from one generation to the next.

3.2 Other further work

In addition to the suggestions in Subsection 3.1 above, other natural next steps would be to try modelling more complex cells with more advanced spindles, eventually building up to cells on the complexity level of mammalian cells, as well as experimenting with more complex artificial mechanisms for DNA partitioning. It would certainly be possible to take small steps that build on this project: for example, the addition of `rescuer` proteins that stop MTs from shrinking was considered, but time constraints meant that this was not properly investigated. In order to accurately model the mitotic spindle in a human cell, for example, more modifications to Cytosim would have to be made, resulting in a more accurate, and hopefully more useful, simulation tool.

4 Methods

4.1 Cytosim

4.1.1 Background

Cytosim[1] is a software suite designed for simulating cytoskeletal dynamics. It is capable of modelling large systems of flexible filaments and associated proteins using a combination of mechanical and stochastic calculations. The Cytosim project was initiated in 1995 by François Nédélec, who is also the current principal maintainer, and it is currently being developed by the Nédélec group at the Sainsbury Laboratory, Cambridge University. The software is run by first compiling the C++ code and then inputting a `.cym` configuration file, which specifies the properties of the system and its components, including

- the time length and time step of the simulation;
- `spaces`, which represent cells;
- objects such as `fibers`, which are linear strings of points, and `beads`, `spheres` and `solids`, which can represent DNA and other molecules;
- `hands`, which can represent proteins (or parts of proteins) with various functions and must be placed in a `single` or `couple` object;
- any events that occur;
- what outputs to print.

Key executable files that can be compiled from the Cytosim code include `sim`, which runs simulations, and `play`, which produces graphics for a simulation either live or using data from a `sim` run.

[1] `https://cytosim.org/`

4.1.2 Main assumptions and techniques

Cytosim makes a number of simplifying assumptions for more efficient simulation (Nedelec and Foethke, 2007). Fundamentally, it uses a fixed time step, agent-based simulation method. At each time point, each component of the simulation (including events and objects) has its `step` function called which determines its state for the next period of time. Points are taken to be zero-dimensional, and `fibers` are one-dimensional sequences of points. As is appropriate for the movements in the cytoplasm, the system is assumed to be at low Reynolds number, meaning inertia is neglected in all calculations. The boundary of the cell is taken to be frictionless, so that the force on any object from the boundary acts perpendicularly to the boundary, and the forces from the boundary are assumed to be linear (Belmonte et al., 2017, Appendix).

Each point-like object experiences Brownian dynamics described by an *overdamped Langevin equation*, $\xi \frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t) + B(t)$, where x is the position vector of the object, ξ is a drag coefficient, $f(x,t)$ accounts for the deterministic forces and $B(t)$ represents a random fluctuation. Multi-point objects such as `fibers` are modelled using points that represent discrete segments. Molecular links are assumed to be Hookean. For objects attached to `fibers`, detachment rates are determined using Bell's law (as outlined in Subsection 1.2), and the speed v of an attached motor under a load is related to its unloaded speed v_0 (as specified in the configuration file) by $v = v_0(1 - f_{\|}/f_m)$, where $f_{\|}$ is the force parallel to the `fiber` and f_m is the specified stall force. Catastrophes for a dynamic `fiber` are modelled using a rate k_{cat} which is related to a specified GTP hydrolysis rate, k_{hyd}, by $k_{\text{cat}} = 3k_{\text{hyd}}^2/k_{\text{gro}}$, where k_{gro} is the growth rate of the `fiber` in (no. of monomer units) s^{-1}. Stochastic events that depend on rates or probabilities are simulated by Gillespie or modified Gillespie approaches.

To solve the Langevin equations at each time step, Cytosim uses *implicit integration*, which has a 1000-fold efficiency improvement over explicit numerical integration.

4.1.3 Modifications

In order to run the simulations in Subsection 2.2 onwards, various additions were made to the base Cytosim code. A `cash` variable was added which stores the total amount of energy, in units of molecules of ATP/GTP, available to the cell, and an new property called `income` was added to

Action	**Name of property**	**Events that require this action**
Synthesis of a `single` or `complex`	`synthesis_cost`	Any attempt to add a `single` or `complex` to the cell
Movement of a `hand` which is attached to a `fiber` by one step	`movement_cost` (per μm)	Any attempt by an attached `hand` (of a `single` or `complex`) to move along a `fiber`
Polymerisation of a `fiber`	`unit_energy_cost` (per μm)	Any attempt to increase the length of a `fiber`, either ‘naturally’ or with assistance from a nucleator

Table 4.1.1: The three energy-costing actions that are accounted for in the simulations in Subsection 2.2 onwards. As with all properties defined in Cytosim, the values of the new properties are specified in the `.cym` configuration file, which makes the simulations highly customisable.

represent the amount of energy gained by the cell per second. It is assumed that, for the simple artificial cell, there are three actions that require a significant amount of energy. These are detailed in Table 4.1.1 together with the names of the new properties that represent the energy costs involved. For each energy-costing event, the wealth of the cell is checked and the event occurs if and only if the cost of the event is not greater than the `cash`.

In addition to the above, for the simulations in Subsection 2.3, extra lines of code were added to the main `step` function, which is called at each time point, that check whether (a) the two DNA `beads` are on opposite sides of the cell (i.e. their x-coordinates have opposite signs) and (b) the absolute values of their x-coordinates are both greater than a certain threshold (hard-coded as 0.5). If both conditions are true, then the DNA is considered to be successfully partitioned, so the program prints the time taken to partition and then restarts the simulation (unless there have been enough replicates). Otherwise, the simulation runs for the specified time length as before.

Code snippets for some of these additions can be found in Appendix A.

4.2 Genetic algorithm

4.2.1 Background

A *genetic algorithm* (GA) was extensively used in this project as the optimisation method for finding parameters that enable the highest success rate for completing a cellular task. This technique is inspired by the process of evolution by natural selection and draws many parallels with it, hence the process by which it works is known as *artificial evolution* (Mitchell, 1998; Rupp and Nédélec, 2012). Given a starting *population* A_0 consisting of sets of randomly generated parameters (within specified bounds) and a *fitness function* f, GAs aim to produce fitter successive *generations* $A_i,\ i \geq 1$. The GA used in this project encodes the values of the parameters as follows: for any *individual* $x_j \in A_0$, the value of each parameter p is encoded as a bit string (a *gene*) whose length is specified. Given specified bounds on p, the value of p is calculated from the bit string via a linear scale. The bit strings for all the different parameters are then concatenated into a *genome* for convenience.

The steps of the GA are detailed in Figure 4.2.1. The intention of the elitism and crossover operations is to create a fitter generation by retaining some of the attributes of the fittest individuals in the current population but also allowing the mixing of genes (or parts of genes) to try to form individuals with the 'best of both worlds'. The mutation operation aims to prevent the GA from converging to a local maximum by introducing perturbations and thus increasing diversity at each bit position. A more rigorous mathematical justification for the validity of GAs, including the *Schema Theorem*, can be found in Chapter 1 of Mitchell (1998).

4.2.2 Implementation

The GA used in this project is written in Python and was originally developed by Maud Formanek and François Nédélec. The author of this report helped to fix bugs and refine the code, as well as writing the code for the fitness functions relevant to this project. The code for the fitness functions can be found in Appendix A.

1. Start with a population A_0 consisting of individuals x_j, where the parameter values for each x_j are encoded in a randomly generated genome.

2. Given the current population A_i, form the offspring population A_{i+1} by calculating the fitness $f(x_j)$ of each $x_j \in A_i$ and then performing the following operations in order:

   (a) *Elitism*: choose an integer $0 < S < |A_i|$, where $|A_i|$ is the population size, and select the fittest S individuals of A_i to be copied into A_{i+1}.

   (b) *Crossover*: choose an integer $0 < P < |A_i| - S$. Select $2P$ individuals, called *parents*, from A_i with replacement using *fitness rank selection*, i.e. the probability of an individual being selected scales linearly as its rank. Now recursively choose two different parents from the selection and form a *child* by taking the first k bits from the first parent and the remaining bits from the second parent. Add all these children to A_{i+1}.

   (c) *Mutation*: choose an integer $0 < Q < |A_i| - S - P$. Select Q individuals from A_i with replacement using fitness rank selection. For each of these individuals, mutate each bit with a specified probability and add the resulting individual to A_{i+1}.

   (d) If at this point $|A_{i+1}| < |A_i|$, complete the new population with random genomes like for A_0.

3. Repeat Step 2 until one of the following conditions is satisfied:

   - the increase in maximum fitness from A_i to A_{i+1} is below a specified threshold;
   - the maximum fitness of an individual in A_{i+1} is above a specified threshold;
   - $i + 1$ reaches a specified upper bound.

Fig. 4.2.1: The steps of the GA used in this project. The first stopping condition in Step 3 was in fact disabled in all the simulations by setting the threshold to 0.

REFERENCES

B. Alberts. *Molecular Biology of the Cell*. Garland Science, Taylor and Francis Group, 2015. ISBN 9780815345244. URL `https://books.google.co.uk/books?id=tLSuoQEACAAJ`.

Hector Aldaz, Luke M Rice, Tim Stearns, and David A Agard. Insights into microtubule nucleation from the crystal structure of human γ-tubulin. *Nature*, 435(7041):523–527, 2005.

Julio M Belmonte, Maria Leptin, and François Nédélec. A theory that predicts behaviors of disordered cytoskeletal networks. *Molecular systems biology*, 13(9):941, 2017.

Christopher S Campbell and R Dyche Mullins. In vivo visualization of type ii plasmid segregation: bacterial actin filaments pushing plasmids. *The Journal of cell biology*, 179(5):1059–1066, 2007.

Alessandro De Simone, François Nédélec, and Pierre Gönczy. Dynein transmits polarized actomyosin cortical flows to promote centrosome separation. *Cell Reports*, 14(9):2250–2262, 2016. ISSN 2211-1247. doi: https://doi.org/10.1016/j.celrep.2016.01.077. URL `https://www.sciencedirect.com/science/article/pii/S2211124716300936`.

Marileen Dogterom and Bernard Yurke. Measurement of the force-velocity relation for growing microtubules. *Science*, 278(5339):856–860, 1997. doi: 10.1126/science.278.5339.856. URL `https://www.science.org/doi/abs/10.1126/science.278.5339.856`.

Colby P. Fees and Jeffrey K. Moore. A unified model for microtubule rescue. *Molecular Biology of the Cell*, 30(6):753–765, 2019. doi: 10.1091/mbc.E18-08-0541. URL `https://doi.org/10.1091/mbc.E18-08-0541`. PMID: 30672721.

Matthew J Footer, Jacob WJ Kerssemakers, Julie A Theriot, and Marileen Dogterom. Direct measurement of force generation by actin filament polymerization using an optical trap. *Proceedings of the National Academy of Sciences*, 104(7):2181–2186, 2007.

Niels Galjart. Plus-end-tracking proteins and their interactions at microtubule ends. *Current Biology*, 20(12):R528–R537, 2010.

Ethan C Garner, Christopher S Campbell, and R Dyche Mullins. Dynamic instability in a dna-segregating prokaryotic actin homolog. *Science*, 306(5698):1021–1025, 2004.

F Gittes, B Mickey, J Nettleton, and J Howard. Flexural rigidity of microtubules and actin filaments measured from thermal fluctuations in shape. *Journal of Cell Biology*, 120(4):923–934, 02 1993. ISSN 0021-9525. doi: 10.1083/jcb.120.4.923. URL `https://doi.org/10.1083/jcb.120.4.923`.

Leland H Hartwell, Joseph Culotti, John R Pringle, and Brian J Reid. Genetic control of the cell division cycle in yeast: A model to account for the order of cell cycle events is deduced from the phenotypes of yeast mutants. *Science*, 183(4120):46–51, 1974.

T E Holy and S Leibler. Dynamic instability of microtubules as an efficient way to search in space. *Proceedings of the National Academy of Sciences*, 91(12):5682–5685, 1994. doi: 10.1073/pnas.91.12.5682. URL `https://www.pnas.org/doi/abs/10.1073/pnas.91.12.5682`.

Hirokazu Hotani and Tetsuya Horio. Dynamics of microtubules visualized by darkfield microscopy: treadmilling and dynamic instability. *Cell motility and the cytoskeleton*, 10(1-2): 229–236, 1988.

J Howard, AJ Hudspeth, and RD Vale. Movement of microtubules by single kinesin molecules. *Nature*, 342(6246):154–158, 1989.

Jacob WJ Kerssemakers, E Laura Munteanu, Liedewij Laan, Tim L Noetzel, Marcel E Janson, and Marileen Dogterom. Assembly dynamics of microtubules at molecular resolution. *Nature*, 442 (7103):709–712, 2006.

Masahide Kikkawa. Big steps toward understanding dynein. *Journal of Cell Biology*, 202(1): 15–23, 2013.

Chapin Korosec and Nancy Forde. Engineering nanoscale biological molecular motors. *Physics in Canada*, 73:78–81, 04 2017.

Akihisa Matsuyama, Ritsuko Arai, Yoko Yashiroda, Atsuko Shirai, Ayako Kamata, Shigeko Sekido, Yumiko Kobayashi, Atsushi Hashimoto, Makiko Hamamoto, Yasushi Hiraoka, et al. Orfeome cloning and global analysis of protein localization in the fission yeast schizosaccharomyces pombe. *Nature biotechnology*, 24(7):841–847, 2006.

M. Mitchell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT Press, 1998. ISBN 9780262631853. URL `https://books.google.co.uk/books?id=gjoiEAAAQBAJ`.

D.O. Morgan. *The Cell Cycle: Principles of Control*. Primers in biology. New Science Press, 2007. ISBN 9780878935086. URL `https://books.google.co.uk/books?id=\_7ygQAOK1DUC`.

Francois Nedelec and Dietrich Foethke. Collective langevin dynamics of flexible cytoskeletal fibers. *New Journal of Physics*, 9(11):427, 2007.

Raja Paul, Roy Wollman, William T. Silkworth, Isaac K. Nardi, Daniela Cimini, and Alex Mogilner. Computer simulations predict that chromosome movements and rotations accelerate mitotic spindle assembly without compromising accuracy. *Proceedings of the National Academy of Sciences*, 106(37):15708–15713, 2009. doi: 10.1073/pnas.0908261106. URL `https://www.pnas.org/doi/abs/10.1073/pnas.0908261106`.

Julien Pernier, Shashank Shekhar, Antoine Jegou, Bérengère Guichard, and Marie-France Carlier. Profilin interaction with actin filament barbed end controls dynamic instability, capping, branching, and motility. *Developmental cell*, 36(2):201–214, 2016.

Beat Rupp and François Nédélec. Patterns of molecular motors that guide and sort filaments. *Lab on a chip*, 12(22):4903–4910, 2012.

Jeanne Salje and Jan Löwe. Bacterial actin: architecture of the parmrc plasmid dna partitioning complex. *The EMBO Journal*, 27(16):2230–2238, 2008.

Virginie Stoppin, Marylin Vantard, Anne-Catherine Schmit, and Anne-Marie Lambert. Isolated plant nuclei nucleate microtubule assembly: The nuclear surface in higher plants has centrosome-like activity. *The Plant Cell*, 6(8):1099–1106, 1994.

Ronald D Vale and Ronald A Milligan. The way things move: looking under the hood of molecular motor proteins. *Science*, 288(5463):88–95, 2000.

Kristen J Verhey, Donna L Lizotte, Tatiana Abramson, Linda Barenboim, Bruce J Schnapp, and Tom A Rapoport. Light chain–dependent regulation of kinesin's interaction with microtubules. *The Journal of cell biology*, 143(4):1053–1066, 1998.

Kristen J Verhey, Neha Kaul, and Virupakshi Soppina. Kinesin assembly and movement in cells. *Annual review of biophysics*, 40:267–288, 2011.

Sam Walcott. The load dependence of rate constants. *The Journal of chemical physics*, 128(21): 06B601, 2008.

Claire E Walczak, Shang Cai, and Alexey Khodjakov. Mechanisms of chromosome behaviour during mitosis. *Nature reviews Molecular cell biology*, 11(2):91–102, 2010.

A Code Snippets

A.1 Modifications to Cytosim

Below are some of the significant blocks of code added to Cytosim.

Additional energy constraints

`simul.cc`

```
⋮
/// The amount of energy the cell has, in units of ATP equivalent
real cash;

/// Checks if the cell has enough 'cash' to perform the action
/// - if so, it returns true and subtracts the passed amount
bool checkWealth(real cost)
{
    if (cash >= cost)
    {
        cash -= cost;
        return true;
    }
    else
    {
        return false;
    }
}
⋮
```

couple_set.cc

```
⋮
void CoupleSet::newObjects(ObjectList& res, const std::string& name, Glossary&
    opt)
{
    CoupleProp * p = simul_.findProperty<CoupleProp>("couple", name);
    Couple * obj = p->newCouple(&opt);

    if ((p->synthesis_cost > 0) && !checkWealth(p->synthesis_cost))
    {
        return;
    }
⋮
```

fiber_site.h

```
⋮
    /// move to a different abscissa on the current fiber
    void moveTo(real a)
    {
        hAbs = a;
        reinterpolate();
    }

    /// moveTo but taking into account energy cost
    void moveTo(real a, real movement_cost)
    {
        if ((a * movement_cost > 0) && !checkWealth(a * movement_cost))
        {
            return;
        }

        moveTo(a);
    }
⋮
```

nucleator.cc

```
⋮
void Nucleator::makeFiber(ObjectList& objs, Simul& sim, Vector pos, std::
    string const& fiber_type, Glossary& opt)
{
    ObjectMark mk = 0;
    Rotation rot(0, 1);

    Fiber * fib = sim.fibers.newFiber(objs, fiber_type, opt);
    Hand const* h = hMonitor->otherHand(this);

    real cost = fib->length() * fib->prop->unit_energy_cost;

    if ((cost > 0) && !checkWealth(cost))
    {
        objs.destroy();
        return;
    }
⋮
```

fiber.cc

```
⋮
    else if ( inc > 0 )
    {
        real cost = inc * prop->unit_energy_cost;

        if ((prop->unit_energy_cost > 0) && !checkWealth(cost))
        {
            return;
        }
⋮
```

Partitioning checkpoint

simul_step.cc

```
⋮
void Simul::step()
{
    #if SPATIAL
        // See if the beads have separated enough - if so, end the simulation
        Bead* dna1 = beads.first();
        Bead* dna2 = dna1->next();
        real dna1_x = dna1->position().x();
        real dna2_x = dna2->position().x();

        if ((dna1_x * dna2_x < 0)
            && std::fabs(dna1_x) > 0.5
            && std::fabs(dna2_x) > 0.5)
        {
            printf("%% dna1_x: %f\tdna2_x: %f\n", dna1_x, dna2_x);
            printf("%% time %f\n", prop.time);
            abortRun = 1;
        }
    #endif
⋮
```

A.2 Fitness functions

arena.py

```python
try:
    import os
    import sys
    import re
    import numpy as np
    from typing import List
except ImportError as e:
    sys.stderr.write("Error loading module: %s\n" % str(e))
    sys.exit()

spatial = False  # set to True if including partitioning checkpoint

# simulation executable must be in current working directory:
simex = os.path.abspath('sim_spatial') if spatial else os.path.abspath('sim')
template = 'config.cym.tpl'
genetic_config = 'genetics.config'
max_time = 100  # units: s


def update_max_time():
    global max_time

    with open(template) as f:
        for line in f:
            if line.strip().startswith('time_step'):
                time_step = float(line.split()[2])
            elif line.strip().startswith('run'):
                max_time = float(line.split()[1]) * time_step
                break


def calculate_fitness(data: List[str], target):
    """
    Calculate fitness expressing how close `data` is to `target`
    Higher fitness is better
    """
    data = data[0].split('\n')

    part_scores = []

    if spatial:
        for i, line in enumerate(data):
            if line.startswith('% report'):
                # Figure out if the partitioning was successful
                if i >= 2 and data[i-2].startswith('% time'):
                    # Partitioning successful
                    t = float(data[i-2].split()[2])
                    score_this_time = 1 - t/max_time
                    # Account for floating point errors
                    score_this_time = np.round(score_this_time, decimals=5)
                else:
                    # Partitioning unsuccessful
                    score_this_time = 0

                part_scores.append(score_this_time)
    else:
        # Remove empty lines
        data_copy = data.copy()
        for line in data:
            if not line:
                data_copy.remove(line)
        data = data_copy

        for i in range(2, len(data), 4):
            dna1 = data[i]
            dna2 = data[i+1]
            dna1_x = float(dna1.split()[2])
            dna2_x = float(dna2.split()[2])

            score_this_time = (-1 * np.sign(dna1_x) * np.sign(dna2_x) + 1) / 2
            # Account for floating point errors
            score_this_time = np.round(score_this_time, decimals=5)
            part_scores.append(score_this_time)

    part_score = np.round(np.mean(part_scores), decimals=5)
    return part_score
```

A.3 Heatmaps and fitness plots

make_fitness_plot.py

```
#!/usr/bin/env python
# M. Yin 2022

"""
Reads the run data from a specified directory and generates a heatmap
based on the distribution of fitness scores for each generation of the run.

Syntax:

    make_fitness_plot.py [directory containing run data]

    For correct naming and labelling, do not end the directory path with a
    '/'.

Example:

    make_fitness_plot.py run0
"""

import os
import sys
```

```python
import matplotlib.pyplot as plt
import numpy as np


spatial = True


# Function for generating heatmap
def heatmap(data, row_labels, col_labels, ax=None,
            cbar_kw={}, cbarlabel="", **kwargs):
    """
    Create a heatmap from a numpy array and two lists of labels.

    Parameters
    ----------
    data
        A 2D numpy array of shape (M, N).
    row_labels
        A list or array of length M with the labels for the rows.
    col_labels
        A list or array of length N with the labels for the columns.
    ax
        A `matplotlib.axes.Axes` instance to which the heatmap is plotted.  If
        not provided, use current axes or create a new one.  Optional.
    cbar_kw
        A dictionary with arguments to `matplotlib.Figure.colorbar`.  Optional
.
    cbarlabel
        The label for the colorbar.  Optional.
    **kwargs
        All other arguments are forwarded to `imshow`.
    """

    if not ax:
        ax = plt.gca()

    # Plot the heatmap
    im = ax.imshow(data, **kwargs)

    # Create colorbar
    cbar = ax.figure.colorbar(im, ax=ax, **cbar_kw)
    cbar.ax.set_ylabel(cbarlabel, rotation=-90, va="bottom")

    # Show all ticks and label them with the respective list entries.
    ax.set_xticks(np.arange(data.shape[1]), labels=col_labels)
    ax.set_yticks(np.arange(data.shape[0]), labels=row_labels)

    # Let the horizontal axes labeling appear on top.
    ax.tick_params(top=False, bottom=True,
                   labeltop=False, labelbottom=True)

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, rotation_mode="anchor",
             size=7)

    # Turn spines off and create white grid.
    ax.spines[:].set_visible(False)

    ax.set_xticks(np.arange(data.shape[1]+1)-.5, minor=True)
    ax.set_yticks(np.arange(data.shape[0]+1)-.5, minor=True)
    ax.tick_params(which="minor", bottom=False, left=False)

    return im, cbar


def main(dir: str):
    """
    Generates a heatmap based on the distribution of fitness scores for each
    generation of the run

    Args:
        dir (str): the directory where all the run data is saved
    """

    if not dir:
        dir = os.getcwd()
    elif dir.endswith('/'):
        dir = dir[:-1]
    dir = os.path.abspath(dir)

    # Get required parameters from evolve.config
    pam = {}
    with open(dir + '/evolve.config', 'r') as f:
        for s in f:
            # Remove commented out lines
            if not s.startswith('%'):
                exec(s, {}, pam)
    pop_size = pam.pop('population_size')
    gen_max = pam.pop('generation_max')
    config_file = pam.pop('config_file')

    # The number of times each simulation is run, as specified by the last
    # line in the config.cym.tpl file
    # e.g. 'restart 9' --> num_reps = 10
    with open(dir + '/' + config_file, 'r') as f:
        for line in f:
            if line.startswith('restart'):
                num_reps = int(line.split()[1]) + 1
                break

    # Extract all the data we need

    x = []  # The generation numbers
    y = []  # The fitness scores
    averages = []  # Average fitness for each generation
    max_fitnesses = []  # Maximum fitness for each generation

    for gen in range(gen_max):
        try:
            with open(f'{dir}/gen{gen:04}/generation.txt') as file:
                scores_for_gen = []

                for l in file:
                    if l == '':
                        break
                    # Add fitness score to list
                    score = float(l.split()[1])
                    y.append(score)
                    scores_for_gen.append(score)
        except FileNotFoundError:
            print(f'Found {gen} gen* directories')
            gen_max = gen
            break

        averages.append(np.round(np.mean(scores_for_gen), decimals=5))
        max_fitnesses.append(np.round(max(scores_for_gen), decimals=5))
        x += [gen] * pop_size

    averages = np.array(averages)
    max_fitnesses = np.array(max_fitnesses)

    fig = plt.figure(figsize=(13, 5))
    ax = fig.subplots()

    # Count the number of occurrences of each fitness score
    num_rows = 21 if spatial else (num_reps + 1)
    upper = 0.02 if spatial else 1

    counts = np.zeros((num_rows, gen_max))
    fitness_scores = np.round(np.linspace(0, upper, num_rows), decimals=5)
    for gen in range(gen_max):
        gen_counts, _ = np.histogram(y[gen*pop_size:(gen+1)*pop_size],
                                     fitness_scores)
        for i, cnt in enumerate(gen_counts):
                counts[-(i+1), gen] = cnt
    fitness_scores = fitness_scores[::-1]

    ax.set_title(f'Heatmap of generational fitnesses for '
                 f'{os.path.basename(dir)}\n')
    heatmap(counts, fitness_scores, range(gen_max), ax,
            cbarlabel='Count', cmap='viridis')
    ax.set_xlabel('Generation')
    ax.set_ylabel('Fitness')

    # Draw average line
    ax.plot(range(gen_max), (upper - averages) * (num_rows - 1), 'w-')
    # Draw max fitness line
    ax.plot(range(gen_max), (upper - max_fitnesses) * (num_rows - 1), 'r-')

    os.chdir(dir)
    filename = f'{os.path.basename(dir)}_heatmap.png'
    plt.savefig(filename)
    print(f'Image saved as {filename}')


if __name__ == '__main__':
    if len(sys.argv) > 1 and sys.argv[1].endswith('help'):
        print(__doc__)
    else:
        main(sys.argv[1])
```

B Configuration Files

The following files were used for the GA runs in Subsection 2.2 with the first evolutionary configuration. The other configuration files were similar.

Cell template file

`config.cym.tpl`

```
% Simulating partitioning

set simul system
{
    time_step = 0.001
    viscosity = 0.1
    display = (label=(Partitioning simulation); size=1200, 700)
    income = 800000
}

set space cell
{
    shape = capsule
}

new cell
{
    radius = 0.4
    length = 2
}

%%% Microtubules
set fiber microtubule
{
    rigidity = 20
    segmentation = 0.2
    confine = inside, 100
    unit_energy_cost = 1625 % 13/0.008

    activity = dynamic
    unit_length = 0.008
    growing_speed = [[growing_speed]]
    shrinking_speed = -0.5
    hydrolysis_rate = 1.0 % catastrophe_rate approximately 1/8.3 s^-1
    growing_force = 1.7
    total_polymer = 200
}

%%% Motors
set hand kinesin
{
    binding_rate = 10
    binding_range = [[binding_range]]
    unbinding_rate = 0.3
    unbinding_force = 2.5

    activity = move
    unloaded_speed = 0.02 % mitotic kinesin very slow
    stall_force = 6
    movement_cost = 125 % 1/0.008

    display = (color=red, red;)
}

set couple complex
{
    hand1 = kinesin
    hand2 = kinesin
    stiffness = 100
    diffusion = 0.05
    specificity = antiparallel
    length = 0.025
    fast_diffusion = 0
    synthesis_cost = 400
}

%%% DNA (plasmids)
set bead dna
{
    confine = inside, 100
    display = (color=green;)
}

%%% Nucleator
set hand nucleator
{
    unbinding = 0, inf % rate, force
    activity = nucleate
    nucleate = 1, microtubule, (length=0.01; plus_end=grow;)
    % rate, name of filament, new microtubule
    display = (color=gray;)
}

set single grower
{
    hand = nucleator
    stiffness = 1000
}

%%% Create objects
new dna
{
    radius = 0.1
    attach = [[int(growers_per_dna)]] grower
    position = -0.5 0 0
}

new dna
```

```
{
    radius = 0.1
    attach = [[int(growers_per_dna)]] grower
    position = -0.3 0 0
}

new event
{
    activity = (new 1 complex)
    rate = [[motor_syn_rate]]
}

run 100000 system
{
    nb_frames = 4
}

report dna:position *
restart 39
```

Evolutionary configuration

evolve.config

```
config_file = "config.cym.tpl"  # name of the config template file
population_size = 50            # size of the population
njobs = 32                      # number of processes
FIT_DIFF_MAX = 0                # convergence criteria on fitness difference
   between two successive generations
FIT_MAX = 0.9                   # convergence on absolute fitness value
elitism = 0.1                   # fraction of population that continues to new
    generation
crossover = 0.7                 # fraction of population that is created from
   mating of two parents
```

```
mutation = 0.2                  # fraction of population that is created from
    mutation of parents
mutation_rate = 0.04            # rate of change per bit of genome
n_plus = 1.3                    # for rank selection, expected number of
    offspring for best individual
resurrect = 1                   # reuse data from old generation when one
    genome reappears
generation_max = 50             # max number of generations to be calculated
```

Parameter ranges

genetics.config

```
growing_speed = ([0.05, 0.5], 8)
binding_range = ([0.01, 0.05], 4)
growers_per_dna = ([1, 12], 4)
motor_syn_rate = ([10, 100], 6)
```