Examensarbete

Modeling and Simulation of Cells

Examensarbete utfört i Information Coding vid Tekniska högskolan vid Linköpings universitet av

Emil Jonsson

LiTH-ISY-EX--11/4365--SE

Linköping 2011
Modeling and Simulation of Cells

Examensarbete utfört i Information Coding
vid Tekniska högskolan i Linköping
av

Emil Jonsson

LiTH-ISY-EX--11/4365--SE

Handledare: Jan-Åke Larsson
ISY, Linköpings universitet

Ingemar Ragnemalm
ISY, Linköpings universitet

Examinator: Robert Forchheimer
ISY, Linköpings universitet

Linköping, 01 November, 2011
The aim of this thesis is to create a computer program that simulates the motion of cells in a developing embryo. The resulting simulator is to be used by the Cell Lineage project (Robert Forchheimer et al.) as an input to their genetic model, the meta-Boolean model [18]. This genetic model is not the focus of this work. Since the simulated system is highly complex, with fluids and deforming soft bodies, it is unfeasible to simulate the system in a physically realistic manner while keeping execution time to reasonable values. Therefore some physical realism is sacrificed in favor of simulation stability and execution speed.

The resulting simulator, Cell-Lab, uses Position Based Dynamics (PBD) [17] to implement a number of different models for the cell’s mechanical properties. PBD is well suited for this purpose since it, while not taking excessively long time to execute, guarantees an unconditionally stable simulation. The simulator includes a hard eggshell surrounding the cells. Cells can be split during the simulation, emulating mitosis. There is also the possibility to simulate cell adhesion using a cadherin like mechanism. To control when and how cells are split and fetch information about the current state of the simulation there is an interface to be used by external applications. The meta-Boolean model can be implemented in such an application.
Abstract

The aim of this thesis is to create a computer program that simulates the motion of cells in a developing embryo. The resulting simulator is to be used by in the Cell Lineage project (Robert Forchheimer et al.) as an input to their genetic model, the meta-Boolean model [18]. This genetic model is not the focus of this work. Since the simulated system is highly complex, with fluids and deforming soft bodies, it is unfeasible to simulate the system in a physically realistic manner while keeping execution time to reasonable values. Therefore some physical realism is sacrificed in favor of simulation stability and execution speed.

The resulting simulator, Cell-Lab, uses Position Based Dynamics (PBD) [17] to implement a number of different models for the cell’s mechanical properties. PBD is well suited for this purpose since it, while not taking excessively long time to execute, guarantees an unconditionally stable simulation. The simulator includes a hard eggshell surrounding the cells. Cells can be split during the simulation, emulating mitosis. There is also the possibility to simulate cell adhesion using a cadherin like mechanism. To control when and how cells are split and fetch information about the current state of the simulation there is an interface to be used by external applications. The meta-Boolean model can be implemented in such an application.
Acknowledgments

I would like to thank my fiancé Sara for putting up with me during the course of this work with all the frustration and joy it has meant for me.

A big thank you goes to my father for encouraging me to finish this report and helping with practical issues around the work.

I would like to thank Philippe Decaudin, Jonathan Dummer, David Stewart for their nice and free C-libraries I’ve used in this work, the free software foundation for GCC and emacs, the Khronos group for OpenGL and Canonical for Ubuntu, being the primary operating system this project has been carried out on.

I would also like to thank my supervisors Jan-Åke Larsson and Ingemar Ragnemalm for their support and help with technical discussions about various ideas, inspiration and tips and tricks.

And finally I would like to thank Robert Forchheimer for giving me the opportunity to participate in the Cell-Lineage project with this small but hopefully helpful part.
## Contents

<table>
<thead>
<tr>
<th>Glossary</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>3</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>4</td>
</tr>
<tr>
<td>1.1.1 Cells</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2 Caenorhabditis Elegans</td>
<td>7</td>
</tr>
<tr>
<td>1.1.3 Cell Lineage Trees and The Cell Lineage Project</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Goal</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Soft Body Simulation</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Different Methods of Soft Body Simulation</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Continuum Methods</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Displacement Fields</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Mass-Spring Systems</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4 Surface Mass-Spring with Pressure Force</td>
<td>13</td>
</tr>
<tr>
<td>2.1.5 Potential Fields</td>
<td>14</td>
</tr>
<tr>
<td>2.1.6 Position Based Dynamics</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Solving Ordinary Differential Equations</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 The Euler Method</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Runge-Kutta Methods</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Verlet Integrator</td>
<td>19</td>
</tr>
<tr>
<td><strong>3 Method</strong></td>
<td>21</td>
</tr>
<tr>
<td>3.1 PBD Applied to Cell Simulation</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1 Relaxing Constraints</td>
<td>22</td>
</tr>
<tr>
<td>3.1.2 Correcting Velocities</td>
<td>23</td>
</tr>
<tr>
<td>3.1.3 Cells</td>
<td>24</td>
</tr>
<tr>
<td>3.1.4 The Environment</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5 Collision Detection and Resolution</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Cell Models</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1 Pair-Pressure Model</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2 Full Body Shape Matching</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3 Surface Shape Matching</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Collision Resolution Constraints</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Cell-Environment Containment</td>
<td>34</td>
</tr>
</tbody>
</table>
Contents

F Using The Simulator 76

Bibliography 78
Glossary

AABB  axis-align bounding box. 25 26

Asymmetric split  One cell splitting into two child cells that are not identical. 3

Barycentric coordinate  An \( N+1 \)-tuple one-to-one parameterization of all points inside of an \( N \)-simplex. The sum of the parameters will is always 1 and all points of the simplex are given by letting the parameters vary between 0 and 1. 38 39

Cadherin  Protein on a cell’s surface that make the cell stick to other cells with the same cadherin on their surface. 4 25 26 34 38 43 54 68 69

Caenorhabditis elegans  Caenorhabditis Elegans, a small well studied roundworm. 4 7 8 25 34 54 59

Chromosome  A structure inside the cell’s nucleus consisting of DNA and proteins. 5 6

CP  controller program. 41 42 45 67 69 73 76 77

Discretization  An approximation of something continuous with a finite set of values. 10 12

DNA  deoxyribonucleic acid. 3 5 6

ECM  extracellular matrix. 7

Embryogenesis  The process by which the embryo is formed and develops, until it develops into a fetus. 7

Eukaryotic  A cell containing organells, such as animal cells. 4

FEM  finite element method. 11

Genome  The set of all genes of a cell. 3 7
Level set surface  An $N$ dimensional manifold defined as all points where a scalar function of $N+1$ variables has a constant value. In this thesis, $N = 2$.

Lineage tree  A structure describing the ancestral relationship between cells in an organism. [3, 4, 7, 8, 41]

Mass-spring system  A system consisting of point masses connected to each other with springs. [12, 14]

Meiosis  A cell is split into four child cells with different sets of genetic information. Used in sexual reproduction. [6]

Meta-Boolean model  A discrete mathematical model of genes and factors in a cell and how they interact. [3, 6, 8]

Mitosis  A cell is split into two child cells with the same genetic information. [6]

ODE  ordinary differential equation. [10, 11, 16, 19]

PBD  position based dynamics. [15, 16, 21, 23, 24, 43, 45, 55, 57, 59, 72]

PDE  partial differential equation. [10, 11]

Polar decomposition  Decomposes a matrix to a product of a symmetric matrix and a rotational matrix. [30, 32]

Prokaryotic  A cell without organelles, such as bacteria. [4]

Shape matching  A soft-body simulation method that matches the current configuration of a body to a rigid transformation of its resting shape and drags the simulated points to the transformed rest shape. [29, 30, 32, 33, 46, 48, 53, 59, 71]

Surface shape matching  Shape matching over overlapping surface patches. [32, 47, 48, 53, 54, 59, 71]
Chapter 1

Introduction

All living organisms consist of cells. Some of one cell, others of many. In the case of multicellular organisms, there are usually many different kinds of cells. The cell’s function, and thereby the organism’s, is determined by its genome. The information of the genome is encoded in the famous deoxyribonucleic acid (DNA) molecule.

Cells multiply by splitting and thereby copying the genetic information inside of them to their children. Therefore, all cells in an organism will have the same genome. The reason different kinds of cells are possible is that special substances (transcriptional activators and repressors) inside the cell controls which parts of the genome is active. A cell split that produces two cells with different activators and repressors gives two cells that will function differently. This is called an asymmetric split.

The Cell Lineage project, of which this thesis is a part, focuses on the process of how one cell through consecutive cell divisions can produce a piece of tissue, an organ or a complete organism. To make this task feasible a highly abstract representation of the cell and its genome has been created. In this representation (called the meta-Boolean model) the genome is modeled with a number of promoter functions acting upon factors in the cell. Factors are boolean in the sense that they are either present of not. Special factors determine when a cell is split, and what different factors to activate in the child cells on an asymmetric split. An example of a promoter function can be written as:

\[ g(A|B, C) \]  

Here, the gene \( g \) will be expressed if the factors \( B \) and \( C \) are present. When expressed, \( g \) will produce the factor \( A \). The notation and mechanics of the meta-Boolean model is described in [18].

All cells with a common ancestor cell (such as the fertilized egg cell of a multicellular organism) can be put in relation to each other in a tree structure called the lineage tree. This tree is interesting since it shows the ancestry of all cells and thus describes when and how the cells specialized to whatever they are now. Using the meta-Boolean model, any cell lineage tree can be produced. The naïve way
of doing this is to give each cell in the tree a corresponding factor and promoter function that split the cells asymmetrically so that the child cells each have their specific factors in them. By examining the wanted lineage tree and identifying similar sub trees this can be improved significantly with respect to the number of promoter functions.

One way to further decrease the number of factors and promoters needed is to let the promoter functions depend on more things than the internal factors. One such possibility is to include some sort of cell to cell communication. This happens in nature either due to chemical signaling or mechanical forces. In both cases knowledge of the cells location with respect to each others is needed to determine the probability of one affecting the other. This is available for some real cell lineage trees; for example, the locations of all cells nuclei in the nematode C. elegans lineage tree are well known and could be plugged into the model.

The focus of my work is on another approach. I will create a physical model for the cells movement and how they interact with each other, then simulate the cells of a lineage tree and use the simulated state as input to the genetic model. The result is a more general method that can be used with other lineage trees than those with known cell locations.

1.1 Background

1.1.1 Cells

In this work I will focus on eukaryotic cells since all multicellular organisms, which is what the cell lineage project targets, have eukaryotic cells. It’s possible to create cell lineage trees for prokaryotic cells (e.g. bacteria) but it wouldn’t be very interesting since prokaryotic cells never differentiate when dividing.

Structure

Limiting the cell is the cell membrane consisting of a lipid bilayer. This membrane is very flexible, yet sturdy enough to keep the cell together. Inside the membrane is the cytoplasm, a water based liquid filled with proteins and organelles. Organelles are small parts inside of the cells devoted to specific tasks, such as converting nutrition to energy (mitochondria), converting light into energy using photosynthesis (chloroplast) and the containing and managing the genetic information (the nucleus). Organelles often have their own lipid bilayer membranes.

Some cells have special proteins on their surface that make cells stick together. These proteins are known as cadherins. There may be more than one type of cadherin proteins in an organism. Cells will only stick to cells of the same cadherin type as themselves. Cadherins is important to cell adhesion.

To maintain the cells shape there is a structure called the cytoskeleton inside the cell. This structure consists of two main components: microtubules and microfilaments. The microtubules are stiff proteins that resist compression. Microfilaments are similar to rubber bands; they resist elongation. Together the tension in the filaments and the resistance to compression in the tubules balances
1.1 Background

Each other and give the cell its shape while allowing it some degree of elastic deformation. This balance of pulling and pushing forces is referred to as a tensegrity structure, originally an architectural term introduced to the field of cell biology by Donald Ingber et al. [13].

Figure 1.1: An example of a tensegrity structure: an icosahedron built from rods and rubber bands. Note that rods are only directly connected to rubber bands and no other rods. Image adopted from Wikipedia

Genes

Inside the cells nucleus is all of the cells genetic information encoded in the double helix shaped molecule deoxyribonucleic acid (DNA). The DNA is organized into multiple chromosomes, each chromosome with 2 identical copies of the genetic material. There are also two of every chromosome, one from the father and one from the mother of the organism in question. The two chromosomes will be slightly different from each others, giving the organism different hereditary traits from its parents.

DNA’s primary function is as a blueprint for all the proteins the cell will produce, and thus for all of the cells’ activities since everything in a cell is controlled by proteins in one way or another. The DNA is divided into genes. Some genes contain information required to produce a protein. A protein is expressed from its gene through an intricate chemical process where the genetic information is copied from the DNA inside the nucleus to a messenger RNA molecule. This messenger
molecule is then transported out of the nucleus to a ribosome where it is translated to the protein.

Not all genes are expressed at all times in all cells. There are many substances that control what genes get expressed in a particular cell. This can work in a variety of ways, from simple enabling factors as in “express this gene only in the presence of this factor” or inhibiting factors as in “don’t express this gene in the presence of this factor” to more complex synergic effects such as “express this gene if factor A and B are present, but not factor C”. This can be emulated in the meta-Boolean model.

Cell division

A crucial feature for any kind of cell is the ability to divide itself into two more or less identical cells. This can happen in two different ways:

In mitosis a cell is split into two cells with all the original chromosomes albeit with only one copy of the DNA per chromosome. After the cell division all the DNA is replicated to get the extra copy of each chromosome. This is the cell division of cells in normal operation that need to multiply.

In meiosis a cell is in two phases split into 4 cells with just one of each chromosome and only one DNA strand per chromosome. The resulting cells are used in sexual reproduction as egg or sperm cells. When an egg cell is fertilized by a sperm cell the resulting cell will have two of every chromosome and can through mitosis grow to become an adult organism of the same kind its parents were.

During meiosis it is not uncommon that something goes slightly wrong. Parts of two corresponding chromosomes can get entangled so that parts of chromosomes change place with each others, i.e. genetic material jumps from one chromosome to the other. This is called chromosomal crossover and is thought to be a driving force of evolution [21].

Cell signaling

Cells can communicate with one another using different methods. This communication typically means that one cell secretes some signaling substance, another cell picks it up and reacts to it e.g. by enabling transcription of some gene. These signaling substances can be transmitted in different ways:

Contact-dependent signaling only occurs between cells that are in direct membrane to membrane contact with each other. Examples of this is the notch-ligand signaling pathway and gap junctions.

Cells can also secrete their signals into the extracellular space and let it diffuse there. This gives slightly further reach than the contact-dependent signaling but still very local.

For more non-local signals nerves and the bloodstream must be used, if present in the organism.
Extracellular structure

In the space between cells there can be an extracellular matrix (ECM). The ECM provides support for the cells and is therefore important to the structural integrity of the organism. The ECM is created by cells secreting special proteins into the extracellular space. These proteins will form the ECM.

Cells can be anchored to the ECM. When they are, the structure of the ECM can affect both the shape of the cell and its ability to communicate with other cells.

The ECM is a very diverse structure with many forms and functions. It is important to understand the ECM in order to understand the structure of any multicellular organism. It is however not present in the early stages of development of C. elegans. Therefore it will not be of focus in this work.

1.1.2 Caenorhabditis Elegans

Caenorhabditis elegans or C. elegans is a small nematode (roundworm). It is frequently used as a model organism in scientific studies because of practical reasons such as that it is transparent, is easy to breed and its embryogenesis is very deterministic and well studied. That is, its cell lineage tree always looks the same, which is very useful for research.

Since C. elegans has been studied so thoroughly many things are known about it that rarely is known about other organisms. It was the first multicellular organism to have its full genome sequenced [1]. The positions of all cells nuclei are known at all times of the embryonal development.

1.1.3 Cell Lineage Trees and The Cell Lineage Project

A cell lineage tree is a structure that describes the ancestral relations between cells in an organism. The root of the tree is the fertilized egg cell. When this cell is divided two new branches will be created in the tree, each representing a new cell.

Given the lineage tree of an organism, some interesting things about its development can be discovered, such as when cells start differentiating into different types of cells.
The Cell Lineage Project aims to find means to reproduce cell lineage trees using a simple mathematical notation called the meta-Boolean model. It uses C. elegans as a model organism due to its predictable cell development and well-studiedness.

1.2 Goal

The aim of this thesis is to create a computer program that simulates the motion of cells in a developing embryo. Since this is a highly complex physical system, with fluids and deforming soft bodies, it might be unfeasible to simulate the system in a physically realistic manner while keeping execution time to reasonable values. In this case some physical realism should be sacrificed in favor of simulation stability and execution speed. The simulation has to be unconditionally stable.

To achieve this a study of different methods of soft body simulations will be performed and a suitable method will be chosen and implemented.

The simulator has to be able to interface with different genetic models. This includes the genetic model dictating which cells split when and where as well as the simulator providing data about the simulation state, such as cell positions and cell contacts, to the model.

The genetic model itself will not be covered by this thesis. This will be a stand-alone simulator able to be used with various genetic models.

A visual representation of the simulation is required. The ability to automatically generate screen shots and video recordings of performed simulations would be useful.
Chapter 2

Soft Body Simulation

In this chapter some different methods of soft body simulation will be investigated.

Soft body simulation handles objects that can change their shape. All objects in the real world are more or less soft bodies. Some are harder than others. Such hard objects are most often simulated using the rigid body approximation since this kind of simulation is well studied and computationally cheap. The rigid body approximation is accurate as long as the objects aren’t hit hard enough to break or deform.

When simulating softer materials, such as biological cells, the rigid body approximation becomes very lacking in realism, so soft body models must be used.

A further categorization is useful: Elastic or non elastic soft bodies. Elastic soft bodies are bodies that, though deformable, will return to their rest shape when the deforming force is removed. Good examples of this are rubber bands and jelly. Non-elastic soft bodies change their shape permanently when exposed to a deforming force. Examples are liquids or clay-like substances.

Some materials are elastic as long as the deformations are small, but become non-elastic when the deformation is big enough. A well known example of this is steel.

Some intuitive observations about the mechanical properties of cells:

- They preserve their enclosed volume.
- They have very flexible membranes, that stretch due to external forces without breaking.
- Some cells have a specific shape they try to maintain in addition to preserving their volume. (e.g. the axon of a neuron)

It appears that an elastic soft body fits these criteria best.
2.1 Different Methods of Soft Body Simulation

2.1.1 Continuum Methods

Continuum methods deal with problems modeled with continuous fields describing the state of the system. A differential equation (usually partial) is defined over the field as well as initial and boundary conditions. When the differential equation is solved, the dynamics of the system are obtained.

A good example of this is modeling the dynamics of water, as in [12]. A partial differential equation (PDE) is defined over the field of water velocities, $u$, and pressure, $p$:

$$\begin{align*}
\frac{\partial u}{\partial t} &= -(u \cdot \nabla)u - \frac{1}{\rho} \nabla p + \nu \nabla^2 u + F \\
\nabla \cdot u &= 0
\end{align*}$$ (2.1)

This particular PDE is known as the Navier-Stokes equations for incompressible flow.

An initial state is defined, giving the initial condition. Various boundary conditions can be used depending on the problem to be solved and the required level of realism.

In the example above, no-slip boundary condition is used for the water velocity and pure Neuman condition for the pressure:

$$\begin{align*}
u(x,t) &= \bar{0}, x \in \partial S \\
\nabla \cdot p(x,t) &= \bar{0}, x \in \partial S
\end{align*}$$ (2.2)

Where $S$ is the set of points for which the system is defined and $\partial S$ its boundary.

To solve the PDE, a discretization is needed in order to approximate solutions to the system with an ordinary differential equation (ODE). In simple cases a regular grid can be used.

When the ODE approximating the PDE is solved over the grid using the boundary and initial conditions realistic water movement is obtained.

This is however a very computationally expensive method. The above article uses the parallelism of modern graphics cards in order to achieve an interactive simulation, but this poses some difficult challenges on the implementation.

2.1.2 Displacement Fields

Displacement field methods are a special kind of continuum methods.

A displacement field describes how every point in a continuum is displaced from its rest state. Using the continuous equivalent of Hooke’s law for springs, forces in every point of the material are obtained. Now, using Newton’s laws of mechanics, we have equations for the acceleration in every point, and can integrate these to new velocities and positions.

Solving this in general requires a discretization just as with the water example above. See [16] for a more detailed description.
In the method described in this article, a point in a material is described by its material coordinate, $x$, and the point’s displacement from its material coordinate, $u(x, t)$. The location of a point in space can thus be represented by world coordinate $p(x, t) = x + u(x, t)$.

The governing equation of an elastic material is Hooke’s law:

$$\sigma = E\varepsilon$$  \hspace{1cm} (2.4)

where $\sigma$ is the stress, $\varepsilon$ is the strain and $E$ is a linear relationship between the two. When comparing this equation to Hooke’s law for a simple one dimensional spring $f = -k \cdot x$ the strain $\varepsilon$ would correspond to $x$, stress $\sigma$ correspond to the force $f$ and $E$ corresponds to the spring constant $k$. But since we are dealing with three dimensional materials in this case, all quantities are tensors instead of scalars.

The strain can be computed from the displacement field either using Green’s non linear strain tensor:

$$\varepsilon_G = \frac{1}{2}(\nabla u + [\nabla u]^T + [\nabla u]^T \nabla u)$$  \hspace{1cm} (2.5)

or by its linearization, the Cauchy linear strain tensor:

$$\varepsilon_C = \frac{1}{2}(\nabla u + [\nabla u]^T)$$  \hspace{1cm} (2.6)

From the strain tensor the stress can be calculated using the $E$ tensor:

$$\begin{bmatrix}
\sigma_{xx} \\
\sigma_{yy} \\
\sigma_{zz} \\
\sigma_{xy} \\
\sigma_{yz} \\
\sigma_{zx}
\end{bmatrix} = \frac{E}{(1 + \nu)(1 - 2\nu)} \begin{bmatrix}
1 - \nu & \nu & \nu \\
\nu & 1 - \nu & \nu \\
\nu & \nu & 1 - \nu \\
1 - 2\nu & & \\
& 1 - 2\nu & \\
& & 1 - 2\nu
\end{bmatrix} \begin{bmatrix}
\varepsilon_{xx} \\
\varepsilon_{yy} \\
\varepsilon_{zz} \\
\varepsilon_{xy} \\
\varepsilon_{yz} \\
\varepsilon_{zx}
\end{bmatrix}$$  \hspace{1cm} (2.7)

where $E$ is Young’s modulus and $\nu$ is Poisson’s ratio. Given the stress tensor the movement of the system is determined by the following PDE:

$$\rho \ddot{\mathbf{p}} = \nabla \cdot \sigma$$  \hspace{1cm} (2.8)

In the general case, solutions to this PDE have to be approximated by solving an ODE obtained by discretizing the material in space and time. For three dimensional materials tetrahedral meshes is the simplest choice. The more detail wanted, the more elements are needed, which requires more computing resources to be executed. This is the basic essence of the ubiquitously used finite element method (FEM).

There are some cases where it is possible to calculate the motion of a displacement field analytically. Small vibrations of a string is one example. This is however not versatile enough to be used for soft body simulation.
2.1.3 Mass-Spring Systems

In a mass-spring system, the particles are attached to each other by elastic springs. An elastic spring has the potential

$$V = \frac{k}{2} \cdot x^2$$ (2.9)

giving the force

$$f = -\frac{\partial V}{\partial x} = -k \cdot x$$ (2.10)

Where $x$ is the spring’s displacement from equilibrium length and $k$ is a positive constant determining the spring’s stiffness. From these forces we acquire accelerations which we integrate to velocities and positions.

For a mass-spring system, the total internal energy can be written as:

$$V = \frac{1}{2} \sum_{i,j:i>j} k_{i,j} (||\mathbf{x}_i - \mathbf{x}_j|| - d_{i,j})^2$$ (2.11)

where $\mathbf{x}_i$ is the position of a point in the system, $d_{i,j}$ is the resting distance between point $i$ and $j$ and $k_{i,j}$ is the spring constant. The force on all points can now be calculated as in (2.10) but using the gradient instead of only one differentiation:

$$\mathbf{F} = -\nabla_X V(X)$$ (2.12)

where:

$$\nabla_X = \sum_{i \in X, j \in X} e_{i,j} \frac{\partial}{\partial x_{i,j}}$$ (2.13)

where $x_{i,j}$ is the $j$ ($j$ is either $x$, $y$ or $z$) component of $\mathbf{x}_i$ and $e_{i,j}$ is the basis vector of $\mathbf{F}$. See appendix A for details on mathematical notation.

Mass-spring systems can be seen as a crude way of modeling Hookian materials. Instead of using the continuous Hooke’s law (2.4) and discretizing this into a solvable system, the already discrete spring equation can be used over a finite set of point masses. The continuous Hooke’s law can be derived by letting the number of point masses in a mass-spring system approach infinity while decreasing the distance between the masses to 0. This gives an intuitive feel for why mass-spring systems can be realistic.

When using mass-spring systems different schemes of connecting the masses together may be used. The most realistic scheme when modeling Hookian materials is to have the points in a grid-like arrangement. Springs attach a mass to other masses close to the first mass. Depending on how many springs are attached to one mass and their spring constants, different material properties can be obtained.

In a mass-spring system like this, where a mass is only connected to a few other masses, most $k_{i,j}$ in (2.11) will be zero. So the system is rather sparse.

When modeling two or three dimensional materials this means that many masses need to be created inside of the material. Especially for three dimensional
objects this can become a problem since the number of masses quickly becomes very high.

To remedy this, three dimensional materials can be modeled using only the point masses on the surface of the object. To preserve the form of the object, springs connecting different parts of the surface have to be created. Since different configurations of these internal springs will give different behavior of the object, this method is less realistic, but requires less computational resources.

If a stiff material is wanted, springs with high spring constant $k$ is needed. This, in conjunction with the usually high number of point masses, gives a very unstable system. To keep the system from exploding, either a combination of damping forces and small simulation time steps or implicit integration methods have to be used such as in [4]. Small time steps increase the cost of the simulation and implicit integration can be difficult to implement efficiently. It is therefore a challenge to implement stiff mass-spring systems in an efficient and robust manner. A trade-off between damping, speed, and robustness is usually required. If large amounts of damping to the system is acceptable, mass-spring systems is an easy way to simulate soft bodies.

Examples of mass-spring systems of different dimensions include:

- A one dimensional string of springs can be used to model rope or hair
- Two dimensional grids of springs can be used to model cloth and other surfaces
- Three dimensional volumes of springs model jelly-like objects

2.1.4 Surface Mass-Spring with Pressure Force

Another way of modeling three dimensional materials using mass-spring systems without any masses inside of the material is to only place masses on the surface of the object, add springs between these and add a force on all points that tries to maintain the volume enclosed by the surface [14]. This is physically similar to a balloon filled with air or water.

The potential energy of the gas inside the surface is defined as:

$$ V_P = K \left( \frac{P_0}{P} - 1 + \ln \frac{P}{P_0} \right) $$

(2.14)

where $P$ is the pressure inside the object and $P_0$ is the surrounding pressure. Differentiating gives the following forces due to pressure:

$$ F_P = -\nabla_X V_P = \ldots = (P - P_0)\nabla_X V $$

(2.15)

where $\nabla_X V$ is the gradient of the volume of the body with respect to all the points in its surface.

This looks straightforward enough. However, if the body is heavily deformed, this force might become very large. This can cause instabilities in the simulation and should be handled. One way of handling it is to introduce another potential
function that gives the same qualitative behavior but avoids the potential risks of \((2.14)\). If the potential is chosen like this:

\[
V_V = \frac{C}{2} (V - V_0)^2
\] 

(2.16)

where \(V_0\) is the resting volume of the object, the following forces are obtained:

\[
F_V = -\nabla X V_V = \ldots = C(V_0 - V)\nabla X V
\] 

(2.17)

which avoids the infinite forces for highly deformed bodies.

Regardless of whether \(V_P\) or \(V_V\) is chosen the volume of the body and its gradient has to be known. The volume of a triangular mesh can be calculated using:

\[
V(X) = \frac{1}{18} \sum_{i \in T} \langle t_{i,1} + t_{i,2} + t_{i,3} \mid (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \rangle
\] 

(2.18)

Where \(t_{i,j}\) is the position of vertex \(j\) in triangle \(i\). See appendix B for more details on this.

### 2.1.5 Potential Fields

The basis for this approach is to, as in the case of mass-spring systems, model the system with a finite set of point masses. Instead of having the fixed structure between the particles given by the springs, all particles can interact with each other. Usually this interaction is described by some potential function between pairs of particles.

This method has been used for a long time in the field of molecular dynamics simulation \([2]\), but have in more recent years gained popularity in modeling other soft materials \([8]\). Particularly fluidic materials such as water or clay can be modeled efficiently using potential fields.

The first step to simulate such a system would be to define a pair potential function:

\[
V_{i,j} = V(r_i, r_j)
\] 

(2.19)

The choice of pair potential gives the qualitative behavior of the system. The potential function must go to zero as the radius between the particles goes to \(\infty\) for the system to be stable without any external stabilizing forces.

The total potential of the whole system is then:

\[
V(X) = \sum_{i \neq j} V_{i,j}
\] 

(2.20)

This gives the force on every particle. Using appropriate numerical integration schemes the evolution of velocities and positions of the particles is obtained.

More complex potential functions not based on a summation of pair-potential functions can also be constructed. Three-body potentials and above are used in
molecular dynamics simulation of materials such as carbon, silicon and germanium [23].

When a fluid is simulated using particles some means of tracking the surface of the fluid must be used to visualize it. To do this, the level set surface of the potential of a test particle, \( \mathbf{r} \), is used. The potential of a test particle is defined as:

\[
V(\mathbf{r}) = \sum_i V(\mathbf{r}, \mathbf{r}_i) \tag{2.21}
\]

The level set of this potential is given by:

\[
\mathcal{S} : \{ \mathbf{r}; V(\mathbf{r}) = c \} \tag{2.22}
\]

For some choice of \( c \). Level set surfaces are also useful in collision detection for soft bodies as the potential function subtracted by the level set constant, \( c \), per definition will have different signs inside and outside of the object.

### 2.1.6 Position Based Dynamics

This approach is based on Position Based Dynamics (PBD) [17]. Unlike the previously mentioned methods this isn’t based on real, physical formulas. Instead, it can be thought of as a geometrical method with dynamics that resemble physical systems.

Position based dynamics (PBD) is usually used to simulate point based systems with stiff constraints between the points, since normal methods have difficulty handling this type of system and PBD excels at it.

To determine the dynamics of a system simulated with PBD constraints are defined over the simulated points. Each constraint has an associated relaxation function that moves the points of the system to a state where the constraint is more fulfilled. There is no guarantee that the constraint will be fully satisfied after just one application of the relaxation function. Many iterations are usually needed.

A PBD time step looks like this:

1. Accumulate external accelerations and accelerations from forces not described with constraints.

2. Predict the state of the system after the time step using symplectic Euler, see (2.32), (2.33).

3. For every constraint, relax the constraint by applying the constraint’s relaxation function in-situ on the system.

4. Repeat the above loop a number of times for increased accuracy.

5. Calculate the next state by introducing forces proportional to the difference between the predicted state before and after the constraint relaxation.
6. Correct the velocities of points that collided during the time step to achieve correct friction and restitution.

NB: The relaxation in step 3 is in the literature referred to as projection as only the part of the state that fulfills the constraint is supposed to remain.

The exact formula used in step 5 is:

\[
V_{n+1} = V_n + \frac{1}{dt} (G - P) + dt \cdot A_{ext} \\
X_{n+1} = X_n + dt \cdot V_{n+1}
\]

Where \( P \) is the predicted state prior to the relaxation and \( G \) is the state after the relaxation. \( A_{ext} \) is the acceleration due to external and non-PBD forces.

Notice that the time step length is explicitly used in the formula. This means that the simulated system’s dynamics will depend on the time step being used. As non-physical as this sounds, it lets us operate directly on the positions of points (e.g. when we project away the constraints) and the velocities of the points will be updated accordingly.

PBD simulators are typically very easy to control and robust, but quite non-physical. For example, the behavior of a heavily constrained system depends more on the number of iterations in step 4 than on the elapsed simulation time, which is not very realistic. Step 6 can also be quite hard to implement reliably.

### 2.2 Solving Ordinary Differential Equations

All of the above mentioned simulation methods include some sort of solving of differential equations. This is not surprising; many phenomena in nature can be described by differential equations, and thus solving them is useful when mimicking nature.

Analytic solutions to differential equations can only be obtained in a few special cases, small vibrations of a string being one case and linear ODE:s being another. But neither of these are versatile enough to be useful for soft body simulation. Therefore numerical methods are needed.

The class of differential equations we will focus on are ordinary differential equations. This is because ODE:s can always be approximated numerically. Partial differential equations can also in many cases be converted to ODE:s making the study of numerical solutions to ODE:s even more important.

An ODE is defined as:

\[
\dot{X}(t) = F(X(t), t)
\]

Where \( t \) is the independent variable (usually time), \( X(t) \) is the state of the system and \( F(X(t), t) \) is the time derivative of the system in state \( X \) at time \( t \).

There exist a large number of methods for solving ODE:s some of which will be described here.
2.2 Solving Ordinary Differential Equations

2.2.1 The Euler Method

The simplest method to solve ODE:s is the Euler method. The method can be derived using a numeric estimation of the time derivative of the state:

$$\dot{X}(t) = \frac{X(t + h) - X(t)}{h}$$  \hspace{1cm} (2.26)

Rearranging this, renaming $X(t)$ to $X_n$, $X(t + h)$ to $X_{n+1}$ and using $F(X(t), t)$ instead of $\dot{X}(t)$ gives the following iteration formula:

$$X_{n+1} = X_n + h \cdot F(X_n, t_n)$$  \hspace{1cm} (2.27)

This is known as the explicit Euler formula. The explicit Euler method is very simple to implement but has very poor precision. Systems simulated using this method typically gain energy every time step due to lack of precision. Especially stiff mass-spring system are very sensitive to this. The energy increase can be mitigated by adding damping to the simulated system.

By using another estimation of the derivative of the state another iteration formula is produced:

$$\dot{X}(t) = \frac{X(t) - X(t - h)}{h}$$  \hspace{1cm} (2.28)

Which gives the following iteration formula:

$$X_{n+1} = X_n + h \cdot F(X_{n+1}, t_{n+1})$$  \hspace{1cm} (2.29)

This is known as the implicit Euler formula. Note that the state at the next time step, $X_{n+1}$ is present both in the left and right side of the identity. Therefore, to obtain the $X_{n+1}$ some method to solve this system is needed, which usually means another iteration function. This makes implicit Euler much more difficult to implement and time consuming to execute than explicit Euler. Implicit Euler doesn’t gain energy like its explicit counterpart which makes it a safer choice when simulating stiff systems.

A third type of Euler integrator is the symplectic, or semi-implicit Euler method. This method is defined for the subset of ODE:s on the form:

$$\dot{x} = f(t, v)$$  \hspace{1cm} (2.30)

$$\dot{v} = g(t, x)$$  \hspace{1cm} (2.31)

Many mechanical systems can be expressed this way. Deriving the iteration formula is more involved for symplectic Euler than for the other types and will not be shown here. The formula is given by:

$$v_{n+1} = v_n + h \cdot g(t_n, x_n)$$  \hspace{1cm} (2.32)

$$x_{n+1} = x_n + h \cdot f(t_n, v_{n+1})$$  \hspace{1cm} (2.33)
Notice that the only difference between symplectic Euler and explicit Euler for this type of ODE is that $v_{n+1}$ is used instead of $v_n$ when calculating $x_{n+1}$, hence the name semi-implicit Euler. However, since $v_{n+1}$ only depends on $x_n$ and time, symplectic Euler is no more demanding to calculate than explicit Euler.

Symplectic Euler, despite the similarities to explicit Euler, is much better at preserving the energy and ensuring a stable simulation of the system. A famous example of this is to simulate a spring. Explicit Euler will be unconditionally unstable whereas symplectic Euler will be conditionally stable.

### 2.2.2 Runge-Kutta Methods

Runge-Kutta methods is a class of numerical solver of ODEs developed by the German mathematicians C. Runge and M.W. Kutta around year 1900. Runge-Kutta methods are important since they describe a wide range of different integration schemes, both explicit and implicit.

A Runge-Kutta method is described by its Butcher tableau:

\[
\begin{array}{c|cccc}
0 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\
0 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\
\hline
& b_1 & b_2 & \cdots & b_s \\
\end{array}
\]

The iteration formula is then:

\[
X_{n+1} = X_n + h \sum_{i=1}^{s} b_i k_i
\]

\[
k_i = F(X_n + h \left( \sum_{j=1}^{s} a_{i,j} k_j \right), t_n + hc_i)
\]

Notice that for \(2.35\) to be explicitly given, $a_{i,j} = 0$ for $j \geq i$, i.e. its Butcher tableau is strictly lower triangular. This corresponds to an explicit Runge-Kutta method. If the Butcher tableau isn’t strictly lower diagonal, it’s an implicit Runge-Kutta method.

Both the explicit and implicit Euler methods are Runge-Kutta methods with the following simple tableaus:

\[
\begin{array}{c|c}
0 & 1 \\
\hline
1 & 1 \\
\end{array}
\]

Explicit \quad Implicit

A commonly used, more advanced Runge-Kutta method is the 4th order Runge-Kutta method, abbreviated \textit{RK4}. The tableau of RK4 is:
This integrator is slightly more demanding to calculate than the explicit Euler method, but has in return much higher numerical accuracy.

### 2.2.3 Verlet Integrator

The Verlet integrator is used to estimate solutions to 2nd order ODEs i.e. ODEs on this form:

\[
\ddot{X}(t) = F(X(t), t) \tag{2.36}
\]

By studying the Taylor expansion of the state \(X(t)\) at time \(t\)

\[
X(t + h) = X(t) + \dot{X}(t)h + \frac{1}{2} \ddot{X}(t)h^2 + \ldots \tag{2.37}
\]

\[
X(t - h) = X(t) - \dot{X}(t)h + \frac{1}{2} \ddot{X}(t)h^2 - \ldots \tag{2.38}
\]

The Verlet integrator can be constructed.

Adding (2.37) and (2.38) together, omitting higher order terms and substituting \(\ddot{X}(t)\) to \(F(\dot{X}(t), t)\) the following iteration formula is obtained:

\[
X_{n+1} = 2X_n - X_{n-1} + F(X_n, t_n)h^2 \tag{2.39}
\]

This integration scheme is highly popular in the field of molecular dynamics simulation due to its simplicity in calculation and relatively high accuracy and stability.
Chapter 3

Method

The product of this work is a 3D cell simulator, Cell-Lab. The previously described framework of position based dynamics (PBD) \[17\] is used, since it offers high stability and flexibility while remaining a relatively computationally cheap alternative.

A number of different cell models have been implemented, each bringing its own constraints into the system. These models will be described in detail here together with a more in-depth explanation of the PBD flavor used in Cell-Lab.

3.1 PBD Applied to Cell Simulation

As stated in section 2.1.6, PBD is a geometrically based method that resembles physical systems. PBD lets you operate directly on the positions of the points in the system giving a high degree of freedom in the development of the simulation.

The most important constituents of a PBD simulation is the state of the system and the constraints that control the dynamics of the time evolution of the state. In Cell-Lab, the state is a set of points with position and velocity vectors. Each point has a specific mass. PBD can handle other primitives than points, but points suit Cell-Lab well. Constraints govern the way the points in the state can move with respect to each other. An example of a constraint is the pair constraint which forces two points to be at a specified distance from each other. Another example is the point–triangle separation constraint which forces a point to be at a specific side of the triangle defined by three other points. Each constraint has a relaxation function that alters the position of the affected points of the state in order to satisfy the constraint.

Below is a more detailed description of a time step in PBD than the one given in section 2.1.6:

1. Accumulate external accelerations and accelerations from forces from interactions not described with constraints (e.g. gravity).

\[
A_n = A_{ext,n} + F_n(X_n, V_n)
\] (3.1)
2. Predict the state of the system after the time step using symplectic Euler.

\[ P_n = X_n + dt \cdot V_n + dt^2 \cdot A_n \] (3.2)

3. Detect collisions between objects in the simulation. When collisions are detected, add collision resolution constraints to the simulation.

4. For every constraint (including collision resolution constraints), relax the constraint by using the constraint’s relaxation function. This can be repeated a number of times to increase accuracy.

\[ G^1_n = P_n \] (3.3)

\[ G^{i+1}_n = R_1(R_2(\ldots R_M(G^i_n)\ldots)) \quad i = 1 \ldots L \] (3.4)

where \( L \) is the number of relaxation iterations, \( M \) the number of constraints applied to the system and \( X' = R_j(X) \) relaxes the system \( X \) into \( X' \) with respect to constraint \( j \).

5. Calculate the next state by introducing forces proportional to the difference between the predicted state before and after the constraint relaxation.

\[ V_{n+1} = V_n + \frac{1}{dt}(G^{L+1}_n - P_n) + dt \cdot A_n \] (3.5)

\[ X_{n+1} = X_n + dt \cdot V_{n+1} \] (3.6)

6. For every collision detected in step 3, correct the velocity of the affected points. This is where friction and restitution is handled. How this is handled is dependent on the type of collision.

### 3.1.1 Relaxing Constraints

In step 4 the relaxation function for every constraint is needed. These can be constructed in a number of different ways.

The most general way of defining a constraint relaxation function is to first define a constraint function. A constraint function is a positive function of the positions of the system’s points. The constraint is considered fulfilled if the constraint function is zero for the given system. To relax a system with respect to a constraint with a constraint function, a gradient descent is performed. The relaxation function is therefore:

\[ X' = R(X) = X - \frac{C(X)}{\|\nabla_X C(X)\|_e^2} \nabla_X C(X) \] (3.7)

where \( C(X) \) is the constraint function and \( \nabla_X C(X) \) its system gradient.

Two important classes of constraint functions are functions that are invariant over rigid transformations and those that aren’t. In the same way that potential functions in mechanics that are invariant over rigid transformations give rise to forces that conserve both total energy and momenta of the system, relaxing a
constraint with a rigid transformation invariant constraint function will conserve the total momenta of the system. The energy however, will not be conserved. See appendix [C] for details on this. Constraint functions that aren’t invariant over rigid transformations, will not conserve momenta when relaxed.

Some types of constraints don’t lend themselves well to the constraint function — gradient descent paradigm. Examples of this is when no useful constraint function can be defined, or the gradient of the constraint function is too complex to be evaluated efficiently. To relax such constraints the points of the system are simply dragged into a position where the constraints are considered fulfilled. As there is no explicit constraint function, some other criteria needs to be defined to evaluate this. No guarantee that relaxing this type of constraints will conserve momenta exists.

Extra care needs to be taken to make sure that all constraint relaxation functions result in a convergent series of states when applied in sequence. For constraints with a constraint function this is fulfilled if the constraint function is convex around all points for which the constraint is fulfilled. This is a sufficient, but non-necessary criteria for convergence. The distance constraint function described in 3.2.1 is convex, but it is quite hard to evaluate this in the general case.

Another sufficient but non-necessary condition for convergence is idempotency of a relaxation function, i.e. when applied in sequence on a state the result will be identical to when it is applied once:

\[ R(R[\ldots R(X) \ldots ]) = R(X) \] (3.8)

This condition is more restrictive than convexity but can also be applied to constraints without constraint functions. If a constraint function is linear, its resulting relaxation function will be idempotent, but using linear constraint functions is quite restrictive.

To control how strictly a constraint is enforced a stiffness factor can be used. The modified relaxation function is defined as:

\[ X' = \tilde{R}(X) = (1 - \eta)X + \eta R(X) \] (3.9)

where \(0 \leq \eta \leq 1\) is the stiffness factor. If many constraints influence the same set of points (which is the normal case in Cell-Lab) in a way where they compete with each other, \(\eta\) (together with the order in which the constraints are relaxed) will determine which constraint takes precedence.

### 3.1.2 Correcting Velocities

The last step, step 6, in the PBD iteration is to correct the velocities of points that collided with something during the time step. This is needed since the constraint solving of collision resolution constraints will not result in the velocity changes anticipated by a collision. The velocity corrections can for instance be used to implement elastic or non-elastic collisions and contact friction.

For example, if a point collides with a wall, PBD collision handling will apply a velocity on the point big enough to move the point out of the wall in one time
step. If an elastic collision was desired, the resulting velocity will not be correct. Thus a velocity correction is needed. The difference is shown in figure 3.1; \( x \) marks the initial position of the point, \( \mathbf{v}_1 \) its initial velocity, \( \mathbf{p} \) its predicted new position, \( \mathbf{g} \) its new position after constraint relaxation and \( \mathbf{v}_2 \) the resulting velocity. In 3.1a \( \mathbf{v}_2 = \mathbf{v}_1 + \mathbf{g} - \mathbf{p} \). In 3.1b \( \mathbf{v}_2 \) is \( \mathbf{v}_1 \) reflected in the plane of the wall.

![Diagram](image)

(a) How a collision is handled without a velocity correction

(b) A velocity correction was used to achieve an elastic collision

Figure 3.1: Velocity corrections at collision resolution

In Cell-Lab, velocity corrections are applied on collisions between points in cells and the surrounding eggshell. The point-shell collisions are fully non-elastic, i.e. the component of the initial velocity along the normal of the shell at the point of intersection is projected away. Friction can also be applied.

### 3.1.3 Cells

Cells are modeled as triangular meshes approximating a sphere to various levels of detail. These meshes are created by starting out with an icosahedron (level of detail = 0) and progressively subdividing each triangle in the mesh to 4 smaller ones and normalizing the radius vectors to any new vertices created on the mesh. This way, the triangles of the mesh are roughly the same size and shape, which is good for stability.

To simplify the development of Cell-Lab, the mass of all points is set to 1. The more correct way would be to distribute the mass of a cell among all the points in its mesh and setting the mass of the cell according to its volume and density. Since cells in Cell-Lab usually have similar volume and identical number of points and density, this simplification doesn’t introduce much error.

To control the movement of the cells different cell models have been investigated. Each cell model brings some specific constraints to the simulation that will be used in the solver in PBD. These cell models will be described in subsequent sections.

Cells can be split. This is performed by inserting two smaller cells (child cells) inside of the splitting cell (parent cell). The child cells increase in size over many time steps until they together reach the total volume of the parent cell. The parent cell is then removed from the simulation and the split is complete. The places the child cells are inserted into the parent determines the split plane.
Cells can have cadherin on their surface. If two cells with the same cadherin type touch each other, they will glue together. This is represented in the simulation by constraints that force the surfaces of the affected cells together.

### 3.1.4 The Environment

The C. elegans egg shell is modeled with an infinitely hard sphere or ellipsoid. There is also an option to completely remove the shell for testing purposes.

Inside this shell the cells move in some liquid solution, which can be simulated using simple viscous damping of the system.

### 3.1.5 Collision Detection and Resolution

There are different kinds of collisions that need different kinds of collision handling in Cell-Lab. These are:

- Cell-Environment containment
- Cell-Cell separation
- Cell-Cell containment

Cell-environment containment is the process that keeps the cells inside the egg shell. Cell-cell separation keep the cells separated and cell-cell containment constraint child cells to being inside their parent cells while a cell split is taking place.

**Cell-Environment Containment**

The cell-environment collisions are handled simply by adding cell-environment constraints to all points of the simulation. This might sound wasteful, since most of the points won’t be close to the shell. However, since determining whether a point lies close enough to the shell to need a cell-environment constraint, costs as much as projecting the constraint, this is not the case.

**Cell-Cell Separation**

To perform collision detection between triangular meshes, every vertex in one mesh has to be tested to determine whether or not it is contained in the other mesh. This is in fact not sufficient, since triangle edges can intersect another mesh without any vertices being contained by it. This is not handled in Cell-Lab since it only creates minor flaws with the highly tessellated (and mostly spherical) geometry used.

Cell-cell separation is the most CPU intensive form of collision handling in Cell-Lab since there are many cells in the simulation and these cells consists of many points that has to be individually handled.

Axis-aligned bounding boxes (AABB:s) are used as a first pass to reduce the work load. An AABB of an object is the intervals over all three coordinate axis that tightly includes all points of the object.
This is very useful in collision detection since two objects whose AABB:s don’t intersect can’t possibly collide and comparing AABB:s to each other is very simple. Thus, as a first pass, AABB:s are found for all cells in the simulation, and only the ones whose AABB:s intersect are examined further.

The rest of the process is dependent on whether the 2 cells with overlapping AABB:s should stick together (which is determined by the cadherin type of the cells) or not. If the cells don’t stick together, we should add a separation constraint. If the cells do stick together, cadherin constraints should be added between the cells.

Regardless if the two cells with overlapping AABB:s stick together or not, we have to determine which points from a cell (A), if any, intersect into the other cell (B). This is done by casting rays from points in cell A against triangles in cell B in the direction $\square_B \rightarrow \square_A$ where $\square_A$ denote the center of the AABB for cell A. This ray casting provides information about:

- Whether the point intersect the other cell or not
- If it does, which triangle the ray hit
- Where on that triangle the ray hit

If the cells stick together, cadherin constraints are created. These constraints will force the point from A to be at the point of intersection with the triangle in B in all subsequent time steps.

If the cells should not stick together, there are two different courses of action. The most accurate option is to add point–triangle separation constraints for every point of cell A that penetrates into cell B to the triangle of cell B given by the ray casting above, see chapter 3.3.3. These constraints will force the point of cell A to be at the right side of the triangle of cell B. These point–triangle constraints are only valid one time step; the ray-casting needs to be performed each step.

The other option is to add plane separation constraints, see chapter 3.3.2. The plane separation constraint simply forces all points from cell A to be at one side of the separation plane and all points from cell B to be at the other. When using separating planes, no ray casting needs to be performed between the cells which can improve performance significantly for cells that don’t stick together. However, only convex contacts to other cells will be possible, which can be too restrictive for some applications.

The used ray casting direction (from the center of cell B to the center of cell A) is an approximation of the more correct procedure of collision detection called swept collision detection. In swept collision detection you don’t just examine the state of the system the way it is after the preliminary time step. You examine how the system moves from the old state to the preliminary state. Thus points becomes lines and triangles becomes volumes and these objects have to be tested against each others.

If swept collision detection isn’t used (as in Cell-Lab) some collisions can be missed. Particularly collisions with small, fast moving objects. Since there are no small, fast objects in Cell-Lab, using non-swept collision detection doesn’t introduce many artifacts.
Cell-Cell Containment

When cells are splitting the child cells must be contained in the parent cell. To accomplish this a triangle-point constraint is created between each vertex of the child cell and the closest triangle of the parent cell.

3.2 Cell Models

A number of different models of a cell’s movement have been implemented.

3.2.1 Pair-Pressure Model

The simplest cell model is the pair-pressure model, based on the works of Matyka and Ollila [14]. This model tries to do two things: maintain the distance between neighboring vertices on the cells surface and maintain the volume of the cell. The result being a cell that conserves its volume and is resistant to stretching forces.

The constraint that conserves distance between two points is a simple distance constraint:

$$C_p(x_1, x_2) = ||x_1 - x_2|| - d_0$$

To constrain the volume of the cell, a function to measure the same is needed, see appendix B:

$$V(X) = \frac{1}{18} \sum_{i \in T} \langle t_{i,1} + t_{i,2} + t_{i,3}|(t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1})\rangle$$

The constraint function following this is:

$$C_V(X) = |V(X) - V_0|$$

To be able to relax these constraints as in 3.1.1 the gradients of the constraint functions must be evaluated.

For the pair constraint:

$$\nabla_X C_p(X) = \nabla_X ||x_1 - x_2|| - d_0$$

$$= \chi \nabla_X ||x_1 - x_2||$$

$$= \chi \nabla_X \sqrt{\langle x_1 - x_2|x_1 - x_2\rangle}$$

$$= \chi \frac{1}{2} \frac{1}{||x_1 - x_2||} 2 \langle x_1 - x_2|\delta x_1 - \delta x_2\rangle$$

$$= \chi \langle n|\delta x_1 - \delta x_2\rangle$$

where
\[ \chi = \begin{cases} 
1 & \text{if } |x_1 - x_2| > d_0 \\
-1 & \text{if } |x_1 - x_2| < d_0 \\
0 & \text{otherwise} 
\end{cases} \]

and

\[ n = \frac{x_1 - x_2}{\|x_1 - x_2\|} \]

so:

\[ \nabla_X C_p(X) = \chi \left( \langle \delta x_1 \rangle \langle \delta x_2 \rangle \left( \frac{|n|}{|n|} - \frac{|n|}{|n|} \right) \right) \quad (3.13) \]

And for the volume constraint:

\[ \nabla_X C_V(X) = |V(X) - V_0| = \chi \nabla_X V(X) \]

\[ \nabla_X V(X) = \nabla_X \left( \frac{1}{18} \sum_{i \in T} \langle t_{i,1} + t_{i,2} + t_{i,3} | (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \rangle \right) \]

\[ = \frac{1}{18} \sum_{i \in T} \left( \langle \delta t_{i,1} + \delta t_{i,2} + \delta t_{i,3} | (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \rangle + \langle t_{i,1} + t_{i,2} + t_{i,3} | \delta t_{i,1} \times (t_{i,3} - t_{i,2}) - \delta t_{i,2} \times (t_{i,3} - t_{i,1}) + \delta t_{i,3} \times (t_{i,2} - t_{i,1}) \rangle \right) \]

Using the following definitions the expression can be simplified:

\[ d_{21} = t_{i,2} - t_{i,1} \]
\[ d_{31} = t_{i,3} - t_{i,1} \]
\[ d_{32} = t_{i,3} - t_{i,2} \]
\[ t_{i,T} = t_{i,1} + t_{i,2} + t_{i,3} \]

into:

\[ \frac{1}{18} \sum_{i \in T} \left( \langle \delta t_{i,1} + \delta t_{i,2} + \delta t_{i,3} | d_{31} \times d_{21} \rangle + \langle t_{i,T} | (c_{d_{32}} \delta t_{i,1} + c_{d_{31}} \delta t_{i,2} + c_{d_{21}} \delta t_{i,3}) \rangle \right) \]

With the entire expression written on matrix form:

\[ \nabla_X C_V(X) = \frac{\chi}{18} \sum_{i \in T} \left( \langle \delta t_{i,1} \rangle \langle \delta t_{i,2} \rangle \langle \delta t_{i,3} \rangle \right) \left( \begin{array}{c}
\langle c_{d_{32}} | t_{i,T} \rangle + |d_{31} \times d_{21} \rangle \\
\langle c_{d_{31}} | t_{i,T} \rangle + |d_{21} \times d_{21} \rangle \\
\langle c_{d_{21}} | t_{i,T} \rangle + |d_{31} \times d_{21} \rangle
\end{array} \right) \quad (3.14) \]

\[ \chi = \begin{cases} 
1 & \text{if } V(X) > V_0 \\
-1 & \text{if } V(X) < V_0 \\
0 & \text{otherwise} 
\end{cases} \]
### 3.2 Cell Models

#### 3.2.2 Full Body Shape Matching

This cell model is based on the technique presented in [15].

**Shape Matching**

In shape matching one tries to find the optimal rigid transformation from the rest state of a number of points to its current state. A rigid transformation consists of a translation and a rotation. These are the quantities that have to be determined to perform shape matching.

Let’s call the points in their current configuration \( p_1, p_2, \ldots, p_n \) and the points in the rest state \( q_1, q_2, \ldots, q_n \).

Define the center of mass for the two sets as:

\[
p_{cm} = \frac{\sum_i m_i \cdot p_i}{\sum_i m_i}
\]

(3.15)

And

\[
q_{cm} = \frac{\sum_i m_i \cdot q_i}{\sum_i m_i}
\]

(3.16)

where \( m_i \) denotes the mass of point \( i \).

The optimal translation \( t \) from rest to current state will thus be:

\[
t = p_{cm} - q_{cm}
\]

(3.17)

Finding the optimal rotation is slightly more involved. A rotation is a linear transformation, so let’s start by finding the optimal linear transformation from rest to current.

It is useful to define the following matrices to describe the current and rest states:

\[
P = \begin{pmatrix}
p_1 - p_{cm} & p_2 - p_{cm} & \cdots & p_n - p_{cm} \\
\vdots & \vdots & & \vdots \\
\end{pmatrix}
\]

(3.18)

\[
Q = \begin{pmatrix}
q_1 - q_{cm} & q_2 - q_{cm} & \cdots & q_n - q_{cm} \\
\vdots & \vdots & & \vdots \\
\end{pmatrix}
\]

(3.19)

We’re looking for the transformation \( A \) that minimizes the error of:

\[
A \cdot Q = P
\]

Minimizing the error of linear systems is solved by the least mean squares method:
This matrix $A$ will generally represent more transformation than a rotation. To find the rotation part of $A$, polar decomposition has to be used:

$$A = R \cdot S$$

where $R$ is a rotation matrix and $S$ is symmetric. The matrix $R$ is the optimal rotation matrix that we seek in shape matching since a symmetric matrix only represents scaling transformation. More details on polar decomposition follow in its own subsection.

Now, both the translation $t$ and rotation $R$ has been determined together representing the optimal rigid transformation from the rest to the current state:

$$g_i = R \cdot (q_i - q_{cm}) + t \quad (3.20)$$
or

$$G = R \cdot Q + T \quad (3.21)$$

where

$$G = \begin{pmatrix} \vdots & \vdots & \ldots & \vdots \\ g_1 & g_2 & \ldots & g_n \\ \vdots & \vdots & \ldots & \vdots \end{pmatrix} \quad (3.22)$$
$$T = \begin{pmatrix} \vdots & \vdots & \ldots & \vdots \\ t & t & \ldots & t \\ \vdots & \vdots & \ldots & \vdots \end{pmatrix} \quad (3.23)$$

The points $g_i$ can be thought of as the goal positions corresponding to $p_i$, i.e. the position $p_i$ should have for the object to not be in a deformed state.

It is difficult to construct any useful constraint function given the goal positions. Therefore, when relaxing a shape matching constraint the affected points are simply moved from their current positions to their goal positions. Using a stiffness factor of 1, the simulated body will behave as a rigid body. If softness is desired, a lower stiffness must be used.

Fully relaxing the shape matching constraint is an idempotent operation so the relaxation function is convergent. According to [15] it also conserves momenta.

Polar Decomposition

The polar decomposition of a matrix $A$ is:

$$A = R \cdot S$$
where $R$ is a rotation matrix

$$\det R \equiv 1$$

$$R^T \equiv R^{-1}$$

and $S$ is symmetric. $S$ can be computed with

$$S = \sqrt{A^T A}$$

The square root of a matrix $B$ is calculated by finding the eigenvalues and vectors of $B$ and performing the square root operation on the eigenvalues:

$$B = V \cdot \Lambda \cdot V^T$$

where

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

so that

$$\sqrt{B} = V \cdot \sqrt{\Lambda} \cdot V^T = V \cdot \begin{pmatrix} \sqrt{\lambda_1} & 0 & 0 \\ 0 & \sqrt{\lambda_2} & 0 \\ 0 & 0 & \sqrt{\lambda_3} \end{pmatrix} \cdot V^T$$

Here the $3 \times 3$ case is shown but it works equally well in other dimensions.

Since $B = A^T A$ in the polar decomposition case $B$ is symmetric and positive semi-definite: $x^T A^T A x = (A \cdot x)^T (A \cdot x) = \|A \cdot x\|^2 \geq 0$. This means that the eigenvalues of $B$ will be real and positive. Thus, the square root can be performed safely. The symmetry also permits usage of highly efficient methods to find eigenvalues.

The rotation matrix $R$ can now be determined:

$$A = R \cdot S = R \cdot V \cdot \sqrt{\Lambda} \cdot V^T \Leftrightarrow$$

$$V \cdot A_V \cdot V^T = V \cdot R_V \cdot \sqrt{\Lambda} \cdot V^T \Leftrightarrow$$

$$A_V = R_V \cdot \sqrt{\Lambda}$$

where

$$A_V = V^T \cdot A \cdot V$$

$$R_V = V^T \cdot R \cdot V$$

When all the eigenvalues are non-zero, the columns of $R_V$ are simply equal to rescaled versions of the columns of $A_V$:
\[ A_V = R_V \cdot \sqrt{\Lambda} = \left( \begin{array}{ccc} \sqrt{\lambda_1} & \cdot & \cdot \\ \cdot & \sqrt{\lambda_2} & \cdot \\ \cdot & \cdot & \sqrt{\lambda_3} \end{array} \right) \iff \left( \begin{array}{ccc} \frac{1}{\sqrt{\lambda_1}} & \cdot & \cdot \\ \cdot & \frac{1}{\sqrt{\lambda_2}} & \cdot \\ \cdot & \cdot & \frac{1}{\sqrt{\lambda_3}} \end{array} \right) \]

\( R_V \cdot \sqrt{\Lambda} = R_V \cdot \sqrt{\Lambda}^{-1} \)

\( \iff R_V = \left( \begin{array}{ccc} \frac{1}{\sqrt{\lambda_1}} & \cdot & \cdot \\ \cdot & \frac{1}{\sqrt{\lambda_2}} & \cdot \\ \cdot & \cdot & \frac{1}{\sqrt{\lambda_3}} \end{array} \right) \) (3.24)

Obviously this requires all the eigenvalues to be greater than zero. If one or more of these eigenvalues are zero the matrix \( B \) is said to be rank deficient. In the simulator this corresponds to points in a 3D structure (rank 3) being deformed to be in one plane (rank 2), line (rank 1) or point (rank 0).

To handle rank deficiency, columns corresponding to non-zero eigenvalues are computed using (3.24) and the reminding columns are set so that

\[ \det R = \det R_V = 1 \]
\[ R_V^T = V^T \cdot R^T \cdot V = V^T \cdot R^{-1} \cdot V = R_V^{-1} \]

is fulfilled. This can be achieved either by performing the cross product between previous columns, or by a Gram-Schmidt process.

The **polar decomposition** of a matrix can also be obtained from singular value decomposition of a matrix \( B \):

\[ B = W \cdot \Sigma \cdot V^T \]

Given this, the **polar decomposition** of \( B \) is:

\[ R = W \cdot V^T \]
\[ S = V \cdot \Sigma \cdot V^T \] (3.25)
\[ S = V \cdot \Sigma \cdot V^T \] (3.26)

But this is computationally more expensive than the eigenvalue decomposition.

### 3.2.3 Surface Shape Matching

The **surface shape matching** cell model performs **shape matching** over many small overlapping regions (referred to as surface patches) on the surface of the cell. This results in more degrees of freedom than full body **shape matching** while restricting the shape of the cell as a whole more than pair-pressure model. A similar approach is presented in [19], but over a regular lattice of points instead of the arbitrarily sized surface patches presented here.

To constrain the cell’s volume, the same volume constraint as in the pair-pressure model is used.
A surface patch is defined as one central vertex, \( p_0 \), on the surface and its neighborhood. A vertex \( p_i \) is a neighbor to \( p_0 \) if the surface distance from \( p_0 \) to \( p_i \) is less than a cutoff radius \( r_c \).

Vertices in a patch are also associated with a weight \( w_i \). The weight of a vertex is a function of the surface distance from \( p_0 \) to \( p_i \). The weighting function should be a monotonically decreasing non-negative function. The Gaussian function or half a period of the cosine function are good examples.

Surface distance is the shortest distance between two points on a surface. In our case the surface is a sphere, so the surface distance is simply the difference in angle between the points multiplied by the radius. In the general case this becomes a shortest path through graph search problem, which is a well studied problem \cite{5}.

Shape matching will be performed on a per patch basis. Define the \( P_i \) and \( Q_i \) matrices for the patch \( i \) corresponding to eqs. (3.18) and (3.19) as:

\[
P_i = \begin{pmatrix}
    \vdots & \vdots & \vdots \\
    p_{i,1} - p_{i,cm} & p_{i,2} - p_{i,cm} & \cdots & p_{i,n} - p_{i,cm} \\
    \vdots & \vdots & \vdots & \vdots
\end{pmatrix} \cdot W_i \tag{3.27}
\]

\[
Q_i = \begin{pmatrix}
    \vdots & \vdots & \vdots \\
    q_{i,1} - q_{i,cm} & q_{i,2} - q_{i,cm} & \cdots & q_{i,n} - q_{i,cm} \\
    \vdots & \vdots & \vdots & \vdots
\end{pmatrix} \cdot W_i \tag{3.28}
\]

where \( W \) is the weighting matrix:

\[
W_i = \begin{pmatrix}
    w_{i,1} \\
    w_{i,2} \\
    \vdots \\
    w_{i,n}
\end{pmatrix} \tag{3.29}
\]

and the patch center in current and resting state

\[
p_{i,cm} = \frac{\sum_j w_{i,j} \cdot p_{i,j}}{\sum_j w_{i,j}} \tag{3.30}
\]

\[
q_{i,cm} = \frac{\sum_j w_{i,j} \cdot q_{i,j}}{\sum_j w_{i,j}} \tag{3.31}
\]

Shape matching constraints can now be created in the same way as in full body shape matching and applied in sequence when projecting the surface shape matching model.

### 3.3 Collision Resolution Constraints

Collision resolution constraints are created by the collision detection system in Cell-Lab. These prevent cells from intersecting, keep child cells inside their parents.
and keep all cells within the environment. These constraint types will be covered here. To read about collision detection, see [3.1.5]

Cadherin' constraints will also be treated here, although they aren’t strictly collision resolution constraints.

### 3.3.1 Cell-Environment Containment

To keep the cells of the simulation in their environment, simple constraints are used. The most common environment in cell-lab is a hard sphere. That is, an impenetrable sphere that all the cells are confined to, corresponding to the egg shell in a real *C. elegans*.

The constraint for the hard sphere can be expressed as:

$$ C_{sh}(X) = \sum_i \max(\|x_i\| - r_0, 0) $$

Note the clamping to 0. This is because this constraint is really of inequality type. The point $x_i$ shouldn’t be projected if not outside of the sphere. Since a constraint function = 0 means that the constraint is satisfied the clamping will have the desired effect.

The egg shell of *C. elegans* however isn’t spherical. It is slightly elliptical. Therefore an ellipsoid shell should be supported as well. The sphere constraint can be generalized to an ellipsoid constraint by introducing a scaling matrix $S$ and applying the spherical constraint to a point transformed by this matrix:

$$ C_{sh}(X) = \sum_i \max(\|S \cdot x_i\| - 1, 0) $$

where

$$ S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} $$

and $s_x$, $s_y$, $s_z$ are where the ellipsoid will cut the $x$, $y$ and $z$ axis respectively.

To use this constraint the gradient of $C_{sh}$ is needed:

$$ \nabla_X C_{sh}(X) = \nabla_X \sum_i \max(\|S \cdot x_i\| - 1, 0) = \sum_i \chi_i \cdot \nabla_X \|S \cdot x_i\| $$

$$ \nabla_X \|S \cdot x_i\| = \nabla_X (\langle Sx_i|Sx_i \rangle) = \nabla_X (\langle x_i|S^2|x_i \rangle)^{\frac{1}{2}} = \frac{\langle \delta_{x_i}|S^2|x_i \rangle}{\sqrt{\langle x_i|S^2|x_i \rangle}} $$

$$ \nabla_X C_{sh}(X) = \sum_i \chi_i \langle \delta_{x_i}|S^2|x_i \rangle \frac{S^2|x_i \rangle}{\sqrt{\langle x_i|S^2|x_i \rangle}} $$

$$ \chi_i = \begin{cases} 1 & \text{if } \|S \cdot x_i\| > 1 \\ 0 & \text{otherwise} \end{cases} $$

In Cell-Lab the shell’s shape is for simplicity reasons always symmetric around the $z$-axis. The shape is determined by 2 parameters: a spherical radius $r$ and
the ratio, \( e \) between the ellipsoid’s radius in the plane \( s_p \) and the radius along the longest axis, \( s_z \). The ratio determines the eccentricity of the ellipsoid. The volume of the shell will be that of a sphere with radius \( r \). The coefficients of the scaling matrix \( S \) is given by:

\[
\begin{align*}
  s_x &= s_y = s_p \quad (3.36) \\
  s_z &= e \cdot s_p \quad (3.37)
\end{align*}
\]

To find \( s_p \) given \( r \) and \( e \) the volume of a sphere of radius \( r \) is set equal to the volume of the ellipsoid given by \( s_p \) and \( s_z \).

\[
V(s_p, s_z) \equiv V(r) \Rightarrow \frac{4\pi s_p^2 s_z}{3} = \frac{4\pi s_p^2 e s_p}{3} = \frac{4\pi e s_p^3}{3} = \frac{4\pi r^3}{3} \Rightarrow e s_p^3 \equiv r
\]

\[
\begin{align*}
  s_p &= e^{-\frac{1}{3}} \cdot r \quad (3.38) \\
  s_z &= e^{\frac{2}{3}} \cdot r \quad (3.39)
\end{align*}
\]

### 3.3.2 Cell-Cell Separator

The cell-cell separator is an approximate but effective way to separate two cells from each other. It originates from the separating axis theorem that states:

**Theorem 3.1** Given two convex objects there exists a line onto which their projections are separate if and only if the objects do not intersect.

This is not directly applicable to our situation since we don’t have convex shapes in the general case, and therefore we cannot always find a separating line even if 2 objects don’t intersect. To overcome this the separating direction is approximated with the direction from one cell’s center to the other’s. Given this direction it is straightforward to check if the cells are separable in this direction or, if they aren’t, see what vertices from one cell intersects into the other and how much.

Of course this falls apart when the cells become non-convex or very non-spherical but convex shapes. It does work convincingly for cells of roughly the same size that are mostly spherical. It detects some false collisions, but doesn’t miss any collision.

The center of a cell is defined as the center of the axis aligned bounding box that surrounds the cells. Let us call the centers of cell A and B and \( \Box_A, \Box_B \). Then we can define the separating axis as:

\[
d_p = \frac{\Box_B - \Box_A}{\| \Box_B - \Box_A \|} = \frac{d_p^0}{\| d_p^0 \|} \quad (3.40)
\]

Then we define the extreme points of the cells, that is, the points of the cells that are closest to the other cell or intersecting farthest:
\[ s_A = \max\{\langle x_i^A - \Box_A d_p \rangle \} \]  \hspace{1cm} (3.41)

\[ s_B = \min\{\langle x_i^B - \Box_A d_p \rangle \} \]  \hspace{1cm} (3.42)

So, if \( s_A \leq s_B \) we definitely don’t have a collision. If not true, we have a collision as far as this algorithm is concerned. It might however, as previously stated, be a false collision.

If a collision is found, we define the mean plane \( \Pi_s \) as the plane that intersects \( d_p \) between \( s_A \) and \( s_B \).

\[ s = \frac{s_A + s_B}{2} \]  \hspace{1cm} (3.43)

\[ \Pi_s : \{x \in \mathbb{R}^3 | \langle x - \Box_A d_p \rangle = s \} \]  \hspace{1cm} (3.44)

It is not useful to define a constraint function for this type of constraint since the gradient of any function depending on \( \Pi_s \) (which the constraint function would do) would be piecewise continuous due to its dependence on the max and min functions. Instead, the relaxation function will simply move points on the wrong side of \( \Pi_s \) to their projection on \( \Pi_s \). This operation is neither idempotent nor momenta conserving, but is rather efficient at separating cells.

### 3.3.3 Point-Triangle Separator

This constraint is used to ensure that one point is on a specific side of a triangle. The triangle is defined by three points. Let us designate the points in a clockwise oriented triangle with \( v_1, v_2, v_3 \) and the single point with \( p \). Then, a good measure of how close the point is to the triangle and on which side is:

\[ d(p, v_1, v_2, v_3) = \left\langle \frac{(v_3 - v_1) \times (v_2 - v_1)}{\| (v_3 - v_1) \times (v_2 - v_1) \|} \right| p - v_1 \right\rangle \]  \hspace{1cm} (3.45)

That is, the distance from the point to the nearest point on the plane. In order to keep \( p \) in front of the triangle, this quantity should always be positive and vice versa. To handle both these cases a quantity \( c \equiv \pm 1 \) is introduced. \( c = 1 \) corresponds to \( p \) being in front of the triangle. A constraint function can now be written:

\[ C_{pt}(X) = \max(-c \cdot d(p, v_1, v_2, v_3), 0) \]  \hspace{1cm} (3.46)

This constraint is used to contain child cells in the splitting cell. It can also be used to separate cells from each other unless plane separation constraints are used instead.

The gradient:

\[ \nabla_X C_{pt}(X) = \nabla_X \max(-c \cdot d(p, v_1, v_2, v_3), 0) = -c \chi \nabla_X d(p, v_1, v_2, v_3) \]
where
\[ \chi = \begin{cases} 1 & \text{if } c \cdot d(p, v_1, v_2, v_3) < 0 \\ 0 & \text{otherwise} \end{cases} \]

\[
\nabla_X d(p, v_1, v_2, v_3) = \nabla_X \left( \frac{(v_3 - v_1) \times (v_2 - v_1)}{\|v_3 - v_1\| \times v_1)} \right| p - v_1 \right) 
\]
\[ = \left( \nabla_X \frac{(v_3 - v_1) \times (v_2 - v_1)}{\|v_3 - v_1\| \times v_2)} \right| p - v_1 \right) + 
\left( \nabla_X \frac{(v_3 - v_1) \times (v_2 - v_1)}{\|v_3 - v_1\| \times v_3)} \right| \nabla_X (p - v_1) \right) \] (3.47)

Derivatives of normalized vectors can be calculated using:
\[
\frac{d}{dt} \frac{v}{\|v\|} = \frac{1}{\|v\|} \left( \frac{d}{dt} v - \frac{d}{dt} \frac{v}{\|v\|^2} v \right) 
\] (3.48)

The following symbols are introduced to make the expression more manageable:
\[ d_{31} = v_3 - v_1 \] (3.49)
\[ d_{21} = v_2 - v_1 \] (3.50)
\[ d_{32} = v_3 - v_2 \] (3.51)
\[ d_{p1} = p - v_1 \] (3.52)
\[ T = d_{31} \times d_{21} \] (3.53)

with the following gradients:
\[
\nabla_X T = c_{d_{32}}^T \delta_{v_1} + c_{d_{31}} \delta_{v_2} + c_{d_{21}}^T \delta_{v_3} 
\] (3.54)
\[
\nabla_X d_{p1} = \delta_p - \delta_{v_1} 
\] (3.55)

The gradient of the constraint function can now be “simplified” to:
\[
\nabla_X C_{pt}(X) = -\frac{\chi c}{\|T\|} \left( \left( c_{d_{32}}^T \delta_{v_1} + c_{d_{31}} \delta_{v_2} + c_{d_{21}}^T \delta_{v_3} - \frac{(c_{d_{32}}^T \delta_{v_1} + c_{d_{31}} \delta_{v_2} + c_{d_{21}}^T \delta_{v_3}) \|T\|}{\|T\|^2} T \right| d_{p1} \right) 
+ \langle T | \delta_p - \delta_{v_1} \rangle \right) 
\] (3.56)

To write this on matrix form, the cross product matrix (see A.8) as well as the projection in plane matrix can be used. The matrix describing a projection in the plane with normal \( n \) is defined as:
\[ P_n = 1 - \frac{|n\rangle\langle n|}{\|n\|^2} \] (3.57)
And so:

$$\nabla_X C_{pt}(X) = -\frac{\chi_C}{\|T\|} (\langle \delta v_1 \rangle \langle \delta v_2 \rangle \langle \delta v_3 \rangle \langle \delta p \rangle) \left(\frac{c_{d3} P_T |d_{p1}| - |T|}{c_{d1} P_T |d_{p1}|} \right)$$

(3.58)

### 3.3.4 Cadherin Constraint

To glue cells together, cadherin constraints are used. The cadherin mechanism in Cell-Lab is very simple; when a vertex of a cell penetrates into another cell that it is supposed to stick to (i.e. the cells have the same type of cadherin on their surfaces), a cadherin constraint is generated instead of a collision constraint. The cadherin constraint is kept until one of the cells are removed from simulation, either by cell death or cell division.

When a vertex from cell $A$ penetrates a cell $B$, an intersection point on the surface of $B$ is found by ray casting from the vertex of $A$ in the direction from the center of $B$ to $A$. The point from $A$ ($p$), the triangle ($t_i$) and the barycentric coordinate ($\beta_1$, $\beta_2$, $\beta_3$) for the intersection point are used in the cadherin constraint:

$$C_{cadh}(X) = ||\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3} - p||$$

(3.59)

where $t_{i,j}$ is the j:th vertex of the i:th triangle. The gradient is needed:

$$\nabla_X C_{cadh}(X) = \nabla_X ||\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3} - p||$$

$$= \left( \frac{\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3}}{||\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3}||} \right) \left( \beta_1 \delta_{t_{i,1}} + \beta_2 \delta_{t_{i,2}} + \beta_3 \delta_{t_{i,3}} - \delta p \right)$$

(3.60)

so, with

$$n = \frac{\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3}}{||\beta_1 t_{i,1} + \beta_2 t_{i,2} + \beta_3 t_{i,3}||}$$

(3.60) can now be written more concisely on matrix form:

$$\nabla_X C_{cadh}(X) = (\langle \delta_{t_{i,1}} \rangle \langle \delta_{t_{i,2}} \rangle \langle \delta_{t_{i,3}} \rangle \langle \delta p \rangle) \left( \frac{\beta_1 |n|}{\beta_2 |n|} \frac{\beta_3 |n|}{-|n|} \right)$$

(3.61)

### Barycentric Coordinates

Barycentric coordinates is a $(N+1)$-tuple of values that form an easy way to describe every point inside an $N$-simplex. We’re interested in barycentric coordinates of the triangle (i.e. $N = 2$).

Let the vertices of a triangle be called $v_1$, $v_2$, $v_3$. Then, any point on the triangle ($p$) can uniquely be expressed as
\[
\begin{align*}
\beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 &= \mathbf{p} \\
\sum_i \beta_i &= 1
\end{align*}
\]

(3.62)

$\beta_1, \beta_2, \beta_3$ are known as the barycentric coordinates of $\mathbf{p}$.

Calculating barycentric coordinates is straightforward:

\[
\begin{align*}
\beta_1 + \beta_2 + \beta_3 &= 1 \Leftrightarrow \\
\beta_1 &= 1 - \beta_2 - \beta_3 \Leftrightarrow \\
\mathbf{p} - \mathbf{v}_1 &= \beta_2 (\mathbf{v}_2 - \mathbf{v}_1) + \beta_3 (\mathbf{v}_3 - \mathbf{v}_1)
\end{align*}
\]

(3.63)

(3.64)

(3.65)

Which, written on matrix form is:

\[
\begin{pmatrix}
\mathbf{v}_2 - \mathbf{v}_1 \\
\mathbf{v}_3 - \mathbf{v}_1
\end{pmatrix}
\begin{pmatrix}
\beta_2 \\
\beta_3
\end{pmatrix}
= 
\begin{pmatrix}
\mathbf{p} - \mathbf{v}_1
\end{pmatrix}
\]

(3.66)

This is a $3 \times 2$ system but since there’s a linear dependence between the vectors $\mathbf{v}_2 - \mathbf{v}_1, \mathbf{v}_3 - \mathbf{v}_1$ and $\mathbf{p} - \mathbf{v}_1$ ($\mathbf{p}$ is in the plane of the triangle) the system is exactly solvable and the solution can be obtained by projecting the vectors on a suitable plane and solving a $2 \times 2$ linear system. A suitable plane is one that is not close to being orthogonal to the triangle’s plane. In Cell-Lab, the coordinate plane with normal closest to the triangle normal is used, since projection on these planes correspond to simply dropping one row from the system.
Chapter 4

Implementation

Cell-Lab is designed to handle the geometrical aspects of cell simulation as described in chapter 3. This includes the movement of cells, splitting cells and cells interacting with each other. Other aspects of the cell simulation such as the genetic model and handling cell lineage trees as discussed in 1.1 is not implemented in Cell-Lab. Instead, Cell-Lab implements a text based protocol that external applications, referred to as controller programs (CP:s), can use to control the simulation and gather information about the current simulation state. The controller program - Cell-Lab interface is described in detail in appendix D. The great advantage of this design is that the CP easily can be replaced without the need to recompile the back end simulator. Also, the CP can be written in any programming language, as long as it complies to the protocol.

The simulator reads a configuration file at start up which specifies some static parameters for the simulator, such as which cell model to use, the level of detail of the cells and the number of relaxation steps for the constraint solver. The configuration file is described in appendix E.

Typical interaction between the simulator and the controller program:

1. CP starts
2. CP starts simulator in separate process
3. Simulator reads configuration file
4. CP sends commands to simulator setting up the initial configuration
5. CP sends 'TICK' command to simulator
6. The simulator performs one time step
7. The simulator sends statistics about the current state to CP
8. CP reads the statistics
9. If some condition is met, quit.
10. CP sends commands to simulator

11. Repeat from 5

Just how the simulator is initiated in 4 and how the CP responds to simulator reports in 8 are the most interesting aspects of the CP.

4.1 The Simulator


The simulator is internally represented by the abstract base class simulator. A few different implementations of this interface were developed during the course of this work, each demonstrating a different aspect of cell simulation. configurable_simulator is the one used in the final product. It incorporates all the functionality described in chapter 3 in a configurable way.

A simulator includes a cell_hierarchy. A cell_hierarchy is a tree structure where all the inner and leaf nodes except for the root are cells. The root node represents the containment of all cells, which can be an egg-shell or free space. The children of the root node are the adult cells in the simulation. Their children (if any) are growing cells inside of a splitting cell.

The simulator also has a state. The state is a vector of the position and velocity of all the points in the simulation. The masses of all points, although not strictly part of the simulation state, are also stored within the state class since the state and the masses are almost always needed together.

There is a class to handle information about which of the simulation points are related to each other and how, connectivity. All points indexed by a connectivity must be a consecutive block in the state. The connectivity will index points in relation from the beginning of this block, so the actual index of a point indexed in the connectivity with $i$ will be $i + offset$.

The state holds a list of attached connectivities and their offsets. To get an actual point, one needs both the connectivity and its associated state. You first have to ask the state for the offset of this connectivity, and then look up the point.

To make this easier, the class geometry was created. A geometry holds a connectivity and a reference to a state the connectivity is attached to. For performance reasons, a geometry also holds the connectivity’s offset in the state. The structure of the state, and therefore the offset of connectivities, might change when cells are inserted or removed from the simulation. Geometries should therefore only be saved for as long as the user can guarantee that the state structure isn’t changed.

To handle the different cell models available, the class structure_params is used. There are different sub classes of this interface, each describing a different cell model. A structure_param is given some parameters relevant to the model.
and a geometry defining the rest state, used to measure lengths between vertices etc, and outputs constraints.

A constraint is an important part of the simulator. The class constraint implements the behavior of a PBD constraint as described in chapter 3.1. The most important functions are functions for projecting the constraint or receiving the gradient of a constraint function.

Constraints can both be outputted by the structure_params class or created by other means, such as collision handling.

Collision detection is handled by a collection of functions corresponding to the operations described earlier in this report. When a collision is detected it’s reported to contact_manager that decides whether to create separation or cadherin constraints based on the cadherin type of the cells. If the cell cadherin types match, the collision point is saved and a cadherin constraint forcing the points to be in the relative configuration will be present in all following time steps.

Cell-Lab provides a graphical view in which the user can inspect the current state of the simulation. It’s also possible to save a picture of the simulation every time step by adding a line in the configuration file. This is useful to create videos of the simulation.
# 4.2 Files

Following is a complete list of the source files of Cell-Lab together with a brief description of their purpose:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.cc</td>
<td>Program entry point and I/O handling</td>
</tr>
<tr>
<td>simulator.hpp/cc</td>
<td>Definition of the simulator interface</td>
</tr>
<tr>
<td>configurable_simulator.hpp/cc</td>
<td>The simulator</td>
</tr>
<tr>
<td>state.hpp/cc</td>
<td>The state</td>
</tr>
<tr>
<td>veclist.hpp/cc</td>
<td>Overloaded operators for lists of vectors</td>
</tr>
<tr>
<td>connectivity.hpp/cc</td>
<td>The connectivity class</td>
</tr>
<tr>
<td>geometry.hpp/cc</td>
<td>The geometry class</td>
</tr>
<tr>
<td>structure__params.hpp/cc</td>
<td>The structure__param base class and several implementations</td>
</tr>
<tr>
<td>constraints.hpp/cc</td>
<td>The constraint base class and several implementations</td>
</tr>
<tr>
<td>collision.hpp/cc</td>
<td>Functions concerning collision detection</td>
</tr>
<tr>
<td>raycasting.hpp/cc</td>
<td>Functions concerning ray casting</td>
</tr>
<tr>
<td>aabb.hpp/cc</td>
<td>Axis aligned bounding box class</td>
</tr>
<tr>
<td>contact_manager.hpp/cc</td>
<td>The contact_manager</td>
</tr>
<tr>
<td>cell_hierarchy.hpp/cc</td>
<td>The cell hierarchy</td>
</tr>
<tr>
<td>cell_factory.hpp/cc</td>
<td>Some classes to create different kinds of cells</td>
</tr>
<tr>
<td>vector.hpp</td>
<td>Vector over real numbers class</td>
</tr>
<tr>
<td>matrix.hpp</td>
<td>Matrix over real numbers class</td>
</tr>
<tr>
<td>operations.hpp/cc</td>
<td>Operations available on vectors and matrices</td>
</tr>
<tr>
<td>utilities.hpp/cc</td>
<td>Various small utility functions</td>
</tr>
<tr>
<td>rendering.hpp/cc</td>
<td>Helper function to render cells</td>
</tr>
<tr>
<td>zpr.hpp/cc</td>
<td>3D camera handling</td>
</tr>
<tr>
<td>conf_file_parser.hpp/cc</td>
<td>Configuration file reader</td>
</tr>
<tr>
<td>glhelper.hpp</td>
<td>OpenGL helper functions</td>
</tr>
<tr>
<td>twb*.hpp/cc</td>
<td>GUI handling</td>
</tr>
<tr>
<td>Makefile</td>
<td>The Cell-Lab makefile</td>
</tr>
</tbody>
</table>
Chapter 5

Results

The result of this work is Cell-Lab, a cell simulator.

Figure 5.1: An example of a few cells simulated in Cell-Lab

Cell-Lab is used to simulate the growth of an embryo inside an egg. The simulation begins with a single cell that multiplies by splitting to any number of cells in the resulting organism.

Cell-Lab is highly configurable through the use of its configuration file and controller program architecture. It’s designed to be completely customizable without the need to recompile its C++ source code (see chapter 4).

The cell simulation is performed using the PBD framework described in [17]. This method is computationally inexpensive and guarantees stability. Three different cell models have been implemented each giving different dynamics to the system.
5.1 Cell Models

In this section the pros and cons of each cell model are evaluated with respect to characteristic dynamics and performance.

5.1.1 Pair-Pressure Model

Cells consist of pair constraints between all adjacent points on the cell’s surface and a volume constraint over all points. Both of these have convex constraint functions which give a convergent relaxation series. The result is a stable cell which conserves its volume and adapts its shape to fill any empty space in its proximity. The cell will not have any resistance to changes in the surface’s curvature.

The execution time needed to relax all constraints in a cell modeled with the pair-pressure model is dependent on the number of points used to model the cell since more points will give pair-constraints.

Below is a picture of a single cell modeled with the pair-pressure model resting on a plane under the influence of gravity. It conserves its volume but not its shape.

![Figure 5.2: A single cell modeled with the pair-pressure model under the influence of gravity](image)

5.1.2 Full Body Shape Matching

Cells consists of a shape matching constraint over the whole cell and and a volume constraint. The shape matching constraint does not have a constraint function, but its relaxation function is idempotent, so these constraints together give a convergent series.

Full body shape matching is quite restrictive for the shape of the cell since all points are dragged towards their global rest positions. To introduce some softness a low stiffness factor must be used. When using low stiffness for the shape matching constraint the volume of the cell will not be conserved. Therefore a volume constraint is needed. Having a higher stiffness of the volume constraint than of the shape matching constraint can introduce heavy artifacts in the cell,
since the low stiffness of the shape matching will let the volume constraint tear
the cell’s surface apart.

Full body shape matching is mostly useful when relatively rigid cells are wanted.
When using high stiffness for the shape matching constraint the cells are well
behaved, but not very soft. Using low stiffness is not recommended.

Performance-wise full body shape matching is very cheap, even for highly tes-
sellated cells. The most expensive part of relaxing this constraint is finding the
eigenvalue decomposition of a $3 \times 3$ symmetric matrix (which is rather inexpen-
sive), regardless of the number of points used to model the cell. This is the best
argument for using this type of constraint.

Below is a picture of a single cell modeled with the full body shape matching
model resting on a plane under the influence of gravity. Although some softness
is introduced by lowering the stiffness of the shape matching constraint, the cell’s
shape is mostly unchanged.

![Figure 5.3: A single cell modeled with the full body shape matching model under
the influence of gravity](image)

5.1.3 Surface Shape Matching

Cells are modeled by many overlapping patches to which shape matching con-
straints are applied. A volume constraint is applied to all points in the cell in
order to conserve volume.

By changing the radius of the surface patches the amount of resistance to
changes of the surface’s curvature can be controlled. Small patches make the cell
behave similarly to the pair-pressure modeled cells, while large patches give rigid
cells, similar to cells modeled with full body shape matching.

A problem with this resistance to changes in curvature and the absence of
self-collision handling in Cell-Lab is that highly deformed cells might end up in
erroneous configurations when relaxed. For instance, cells that have “rolled into”
themselves will after relaxation only roll more into themselves since this, locally,
satisfies the surface shape matching constraint better than rolling out to the correct
correct configuration. Cells modeled with the pair-pressure model can also roll into
themselves, but since surface shape matching introduces resistance to changes in curvature of the surface this model is more susceptible to these artifacts.

To counter these degenerate cases a full body shape matching constraint with very low stiffness is also applied to a cell modeled with surface shape matching. In normal configurations this shape matching constraint will not give any noticeable effect due to its low stiffness, but it will help degenerate cells to return to their correct shapes.

Cells modeled with surface shape matching primarily consist of many patch shape matching constraints and a volume constraint. Each of these constraints’ relaxation functions are convergent. However, when applied in sequence after each other they do not give a convergent series of states. This shows up as oscillatory “jittering” motions of the modeled cell. This is most noticeable with highly tessellated cells and small surface patches. The oscillations are bounded in magnitude.

The performance of surface shape matching is highly dependent on the tessellation of the cell and the size of the surface patches. It is generally much more computationally expensive than both the pair-pressure and full body shape matching cell models.

See figure 5.5 for examples of how single cells behave for different $\sigma$-values (surface patch sizes) in the surface shape matching model. Notice the irregular shape of 5.5a caused by the “jittering” described above.
Figure 5.5: Cells modeled with the surface shape matching model with different σ-values under the influence of gravity
5.2 Collision Detection

Two different methods of cell separation have been implemented, the plane- and the point–triangle separator. The plane separation method is much cruder than the point–triangle separation method, but is in return faster to execute. Plane separation introduces heavy artifacts to the simulation due to its inability to handle non-convex contacts between cells. However, in situations with many cells of the same size, the cells will have mostly convex contacts to other cells. In this case plane separation collision handling can be a good alternative.

Below is a comparison of how the two different cell separation methods behave in small systems. Point–triangle separation allows the cells to fill the space between them much more efficiently than the plane separator since non-convex contacts can be handled by point-triangle separation.

![Figure 5.6: An embryo in the 4-cell stage simulated with different types of cell separation methods](image)

(a) Plane separation  
(b) Point-triangle separation

Figure 5.6: An embryo in the 4-cell stage simulated with different types of cell separation methods
5.3 Performance

Below are 2 tables of timings of simulator runs. All parameters are identical in both runs except for the number of relaxation steps performed; 1 in the first table, and 3 in the other.

<table>
<thead>
<tr>
<th>Number of cells</th>
<th>Time per frame (ms)</th>
<th>Time per cell and frame (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>76</td>
<td>9.5</td>
</tr>
<tr>
<td>32</td>
<td>188</td>
<td>5.9</td>
</tr>
<tr>
<td>128</td>
<td>489</td>
<td>3.8</td>
</tr>
<tr>
<td>512</td>
<td>2027</td>
<td>3.9</td>
</tr>
</tbody>
</table>

(a) 1 relaxation step

<table>
<thead>
<tr>
<th>Number of cells</th>
<th>Time per frame (ms)</th>
<th>Time per cell and frame (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>100</td>
<td>12.5</td>
</tr>
<tr>
<td>32</td>
<td>270</td>
<td>8.4</td>
</tr>
<tr>
<td>128</td>
<td>1034</td>
<td>8.0</td>
</tr>
<tr>
<td>512</td>
<td>4615</td>
<td>9.0</td>
</tr>
</tbody>
</table>

(b) 3 relaxation steps

Table 5.1: Performance benchmark

Apparently, except for some constant overhead per frame, execution time scales approximately linearly with respect to both number of cells and the number of relaxation steps used. This is expected and well behaved.
Chapter 6

Discussion

In this chapter some issues with the current solution is presented. A number ideas for future improvements are also discussed.

6.1 Issues

6.1.1 Momenta Drift

Some constraints in a simulation might introduce small amounts of momenta when projected due to the finite precision of the computer or non conservative relaxation functions. Since the geometry of the egg-shell is symmetric around one axis these small amounts of momenta can lead to a global momenta build up in the simulation in the form of a tumbling motion around the eggs symmetry axis. Friction against the egg shell helps preventing this.

6.1.2 Degenerate Shapes of Cells

As described in section 5.1.3 cells (not only those modeled with surface shape matching) might reach degenerate shapes when returning from a heavily deformed shape. This is indeed a well known problem when simulating soft bodies.

In Cell-Lab this can be prevented to some degree by adding a full body shape matching constraint with low stiffness over the whole cell to the cell model. This helps against large defects such as when a cell is “rolled into itself” as in fig. 5.4 but not against smaller defects. A typical smaller defect can be observed when cells slide against the shell of the egg. The friction between the shell and cell can force some of the cells surface to fold into itself. This deformation is not big enough to be prevented by a full body shape matching constraint. To prevent this, proper self collision handling must be used.
6.1.3 Jittering of Surface Shape Matching

See 5.1.3 Surface shape matching introduces jittering of the cell’s shape. This is because the surface shape matching constraint is not necessarily convergent.

6.2 Future Work

6.2.1 More Advanced Visualization Options

Cell-Lab includes some basic visualization of the simulation. The user can view the current state of the system and change the “camera’s” position.

This allows the user to inspect what happens to the cells closest to the egg shell, but hard to follow what goes on inside of the embryo. As long as only a few cells are simulated they will all be in contact with the egg shell and therefore be visible. But when more cells are simulated some other mechanism is needed to inspect the state of all cells. This could be as simple as an option to hide some of the cells so that others become visible. Better yet would be to be able to render cross-sections of the simulated state. This is a particularly attractive solution since such cross-sections are available for real embryos of *C. elegans* [20], and could therefore be used to verify the validity of the simulation.

Many other things could be improved in the visualization to make the simulation look prettier such as smoother shading of the cells, indicators of different types of contacts between cells (separation or glued together by cadherin) and nicer transitions when cells split. Some visual representation of the egg shell would also improve the visual quality of the simulator.

6.2.2 Better Curvature Preserving Cell Model

The surface shape matching cell model was introduced in order to let cells preserve their surface curvature in addition to the conservation of lengths and angles that the pair–pressure model handles. This can be introduced in other ways, possibly without the jittering issue. A simple example of such a cell model is to add bend–resistance constraints to the pair–pressure model. For every pair of adjacent triangles with normals $\mathbf{n}_1$ and $\mathbf{n}_2$ and normalized vector along the edge connecting the cells $\mathbf{v}_{1,2}$ the following constraint function would preserve the angle between the triangles:

$$C(X) = \| \cos \theta^0 \cdot \mathbf{n}_1 + \sin \theta^0 \cdot \mathbf{n}_1 \times \mathbf{v}_{1,2} - \mathbf{n}_2 \|^2$$  \hspace{1cm} (6.1)

where $\theta^0$ is a constant chosen so that $C(X) = 0$ in the resting configuration of the system, that is:

$$\cos \theta^0 \cdot \mathbf{n}_1^0 + \sin \theta^0 \cdot \mathbf{n}_1^0 \times \mathbf{v}_{1,2}^0 \equiv \mathbf{n}_2^0$$  \hspace{1cm} (6.2)

$\mathbf{n}_1$ and $\mathbf{n}_2$ will always be orthogonal to $\mathbf{v}_{1,2}$ so (6.2) is exactly solvable.
6.2 Future Work

6.2.3 Momenta Compensation

To give the user of Cell-Lab some more freedom to implement constraints that
don’t conserve momenta a momenta conservation mechanism could be developed.
Using this, the user could implement a constraint with a relaxation function that
doesn’t conserve momenta, and then let the momenta conservation mechanism
add velocities to the system in a way that subtracts the inserted momenta using
the velocity correction mechanism.

6.2.4 Self Collisions

Cells in the current incarnation of Cell-Lab are only collision tested against other
cells. In order to prevent cells from intersecting themselves self collisions need to
be tested as well. This is hard to implement and expensive to execute. [3] provides
a method that would prevent the self intersections that cause degenerate shapes
in Cell-Lab.

6.2.5 Parallelization

To improve the performance of Cell-Lab the simulation algorithm could be adapted
to parallel execution to utilize the processing power of multi-core processors which
are common today.

The PBD approach used in Cell-Lab does not lend itself well to parallelization
in its basic form. The constraint solver is an iterative process where each iteration
depends on the result of the one before it (see 3.1).

An easy adjustment to this algorithm would simply be to split the constraint
list into one part per participating execution unit. This algorithm would look
something like this (it would replace step 4 in the current implementation):

1. Create one copy of the state for each execution unit
2. Relaxation is performed in parallel on each execution unit on its own state
3. The updated state is calculated as the mean value of all the copies of the
   state
4. Repeat steps 1 to 3 until sufficient precision is obtained

Further steps in parallelization would be to adapt the program for execution
on a massively parallel architecture such as graphics cards. Major changes to the
algorithm would be needed for this however.

6.2.6 Other Constraint Solution Schemes

Other types or constraint solvers can be implemented and evaluated. An interest-
ing way forward would be to implement a method to solve all constraints simulta-
neously instead of sequentially as now. Solving sequentially works well for systems
with fewer constraint functions than degrees of freedom but can give problems
with convergence on overly constrained systems.
To solve all constraints simultaneously one would define a constraint vector (it is assumed that all constraints have constraint functions):

\[
\mathbf{C}(X) = \begin{pmatrix}
    C_1(X) \\
    C_2(X) \\
    \vdots \\
    C_m(X)
\end{pmatrix}
\] (6.3)

The system fulfills all constraints when \( \mathbf{C}(X) = 0 \). To be able to solve this equation \( \mathbf{C}(X) \) is linearized around \( X \):

\[
\mathbf{C}(X + \Delta X) = \mathbf{C}(X) + \nabla_X \mathbf{C}(X) \Delta X = 0 \Rightarrow \nabla_X \mathbf{C}(X) \equiv J(X) \Rightarrow
\]

\[
J(X) \Delta X = -\mathbf{C}(X)
\] (6.4)

Where \( J(X) \) is referred to as the Jacobian matrix.

This needs to be solved for \( \Delta X \). Depending on whether the number of constraint functions, \( m \), is bigger or smaller than the number of degrees of freedom of the system, \( n \), different methods need to be used. Below is a summary of the those methods. All matrices are assumed to have full rank which in reality might not be true.

\( m < n \) The system is under-constrained. This is the normal case in Cell-Lab and other [PBD] simulators. (6.4) is solved by setting \( \Delta X \) to a linear combination of all the gradients of the individual constraint functions:

\[
\Delta X = J^T \Lambda
\] (6.5)

Where \( \Lambda \) is a vector of mixing coefficients. (6.4) becomes:

\[
J J^T \Lambda = -\mathbf{C}
\] (6.6)

This equation is solvable for \( \Lambda \) since \( J J^T \) is an \( m \times m \) matrix. Note that this method simplifies to the method used in section 3.1.1 in the case of one constraint function. Also note that since \( \Delta X \) is a linear combination of gradients of constraint functions no momenta will be introduced when moving the system in the direction given by \( \Delta X \) if all constraint functions are invariant over rigid transformations for the same reasons as in appendix C.

\( m = n \) The system is exactly solvable. (6.4) can be solved directly. This means that the system is forced to be in one exact configuration, since the number of constraint functions is equal to the number of degrees of freedom of the system. This is not the common case in simulations; things are usually allowed to move!
$m > n$ The system is over-constrained. The least mean squares method can be used to find a solution that is closest to satisfying all the constraints:

$$J^T J \Delta X = -J^T C$$  \hspace{1cm} (6.7)

All the above methods lead to numerical solving of large matrices, something which is rather expensive but lends itself better to parallel execution than ordinary PBD [10].
Chapter 7

Conclusions

It has been shown that position based dynamics is a viable way of simulating cells of a growing embryo inside a fertilized egg. Using the controller program protocol any arrangement of cells can be achieved by choosing when cells split and in which direction. In my work with this thesis I have only written simple “dummy” controller programs that showcase some of the simulators properties or measure the performance of the simulation. My supervisor, Jan-Åke Larsson has created a controller program that accurately simulates the first seven cell splits of *C. elegans* But further studies are needed in this area to prove that cell simulation is a valuable tool for the Cell Lineage project.

Out of the three presented cell models, surface shape matching provides the most versatile dynamics, while the pair–pressure and full body shape matching models don’t introduce the jittering that the surface shape matching modelled cells suffer from. It is the authors recommendation to use the pair–pressure model whenever its dynamics are deemed sufficient since the surface shape matching model takes much more execution time to use and introduces its own problems in addition to its improved dynamics.

PBD in general is very stable and reliable but has some other draw–backs that are hard to neglect. Performing gradient descent on constraint functions may result in huge movements in the affected points since the points are simply moved to a position where the constraint is fulfilled (regardless of how big velocity this displacement represents). This is partially what is responsible for the degenerate cell shapes described in section 5.1.3. Another problem is that in a system with lots of contacts and interactions, such as the cell clusters usually simulated in Cell-Lab, the resulting dynamics will depend more on the number of performed relaxation steps than the elapsed time of the simulation. But, as was stated in section 1.2 stability and execution time is more important in this work than physical accurateness.
Appendix A

Mathematical Notation

This work includes a lot of vector mathematics and differentiation of complex (as in non simple, not with imaginary components) functions. To avoid any ambiguity the used notation will be presented and discussed in this appendix.

A geometric vector is a vector in three dimensions. Since Cell-Lab is a three dimensional simulator of cells, geometric vectors are of considerable importance. Geometric vectors are written in bold:

\[ \mathbf{x}_i = x_{i,x} \hat{x} + x_{i,y} \hat{y} + x_{i,z} \hat{z} \]  
(A.1)

The scalar product between two vectors uses bracket notation:

\[ \langle \mathbf{x}_i | \mathbf{x}_j \rangle = x_{i,x}x_{j,x} + x_{i,y}x_{j,y} + x_{i,z}x_{j,z} \]  
(A.2)

When calculating scalar products of vectors that are transformed by matrices, the matrix can be put in the middle of the product as it doesn’t apply to one vector or the other as much as to the whole product:

\[ \langle \mathbf{x} | A \mathbf{y} \rangle = \langle \mathbf{y} | A^T \mathbf{y} \rangle \]  
(A.3)

\[ \langle A \mathbf{x} | \mathbf{y} \rangle = \langle \mathbf{x} | A^T \mathbf{y} \rangle \]  
(A.4)

\[ \langle A \mathbf{x} | B \mathbf{y} \rangle = \langle \mathbf{x} | A^T B \mathbf{y} \rangle \]  
(A.5)

\[ \langle \mathbf{x} | A \mathbf{y} \rangle = \langle \mathbf{y} | A^T \mathbf{x} \rangle \]  
(A.6)

The cross product is another important operation between two geometric vectors:

\[ \mathbf{x} \times \mathbf{y} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ x_x & x_y & x_z \\ y_x & y_y & y_z \end{vmatrix} = -\mathbf{y} \times \mathbf{x} \]  
(A.7)

Performing the cross product between a vector \( \mathbf{a} \) and any other vector \( \mathbf{x} \) can also be expressed as a matrix multiplication:
\[
\mathbf{a} \times \mathbf{x} = \mathbf{c}_a \mathbf{x} \quad \text{(A.8)}
\]
\[
\mathbf{c}_a = \begin{pmatrix}
\mathbf{a} \times \dot{\mathbf{x}} & \mathbf{a} \times \dot{\mathbf{y}} & \mathbf{a} \times \dot{\mathbf{z}}
\end{pmatrix} \quad \text{(A.9)}
\]
\[
\mathbf{c}_a = -\mathbf{c}_a^T \quad \text{(A.10)}
\]

To represent the configuration of the system two system vectors are used, one for positions of the points and one for the velocities:

\[
\mathbf{X} = e_{x_1,x} x_{1,x} + e_{x_1,y} x_{1,y} + e_{x_1,z} x_{1,z} + \ldots + e_{x_N,x} x_{N,x} + e_{x_N,y} x_{N,y} + e_{x_N,z} x_{N,z} \quad \text{(A.11)}
\]
\[
\mathbf{V} = e_{v_1,x} v_{1,x} + e_{v_1,y} v_{1,y} + e_{v_1,z} v_{1,z} + \ldots + e_{v_N,x} v_{N,x} + e_{v_N,y} v_{N,y} + e_{v_N,z} v_{N,z} \quad \text{(A.12)}
\]

where \(e\) are the basis vectors and \(N\) is the number of points in the system.

It’s important to understand the relationship between system and geometric vectors; the system position vector contains the position of all points in the system. Given the system position vector, \(N\) geometric vectors can be constructed each pointing at the position of one point. The system vector represents the entire system, while a geometric vector only represents a point in three-space.

Scalar products can be taken between two system vectors. This is less common than scalar products between geometric vectors. If not obvious from context a system vector scalar product is distinguished from a geometric with a small \(e\) within the bracket. This is normally omitted.

\[
\langle \mathbf{X} | e \mathbf{Y} \rangle \quad \text{(A.13)}
\]
\[
\langle \mathbf{X} | e \mathbf{X} \rangle = ||\mathbf{X}||_e^2 \quad \text{(A.14)}
\]

The nabla operator with respect to a system vector is defined as:

\[
\nabla_\mathbf{X} = \sum_{i=1}^{N} e_{x_i,j} \frac{\partial}{\partial x_{i,j}} \quad \text{(A.15)}
\]

which is used to produce system gradients of functions.

A special and useful vector that is encountered when differentiating functions of a system vector expressed using geometric vectors is \(\delta_{\mathbf{x}_i}\). \(\delta_{\mathbf{x}_i}\) is the system gradient of the geometric vector \(\mathbf{x}_i\):

\[
\delta_{\mathbf{x}_i} \equiv \nabla_\mathbf{X} \mathbf{x}_i = e_{x_i,x} \hat{x} + e_{x_i,y} \hat{y} + e_{x_i,z} \hat{z} \quad \text{(A.16)}
\]

An example of the usage of \(\delta_{\mathbf{x}}\) is when differentiating the squared distance of a point from the origin.
\[ C(X) = ||x||^2 = \langle x|x \rangle \]

\[
\nabla_X C(X) = \langle \nabla_X x|x \rangle + \langle x|\nabla_X x \rangle \\
= \langle \delta_x|x \rangle + \langle x|\delta_x \rangle \\
= 2\langle \delta_x|x \rangle \\
= 2(e_{xx}x_x + e_{xy}x_y + e_{xz}x_z) \\
\]

A more complex example is the system gradient of the triple product, written on matrix form:

\[
C(X) = \langle a|b \times c \rangle \\
\nabla_X C(X) = \langle \delta_a|b \times c \rangle + \langle a|C_b^\delta c \rangle \\
= \langle \delta_a|c_b^c \rangle + \langle a|c_b^T \delta c \rangle + \langle a|c_b^c \delta c \rangle \\
= \langle \delta_a|c_b^c \rangle + \langle \delta_b|c_b^a \rangle + \langle \delta_c|c_b^T \rangle \\
= (\langle \delta_a| \langle \delta_b| \langle \delta_c| \begin{pmatrix}
  c_b^c \\
  c_b^a \\
  c_b^T
\end{pmatrix}
)
Appendix B

An Expression for the Volume of a Triangular Mesh

A triangular mesh defines a surface of an object. To calculate the volume of the object, some expression is needed that takes the triangle vertex positions as input and produces a volume as output.

Gauss theorem states that for some function \( F(r) \) and surface \( S \) enclosing the volume \( V \):

\[
\int_S \langle F(r)|\hat{n} \rangle \, dA = \int_V \langle \nabla|F(r) \rangle \, d\tau \quad \text{(B.1)}
\]

Then, by choosing \( F(r) \) in a suitable way, the right hand side of this expression will be a constant times the volume \( V \). One such choice is \( F(r) = r \). The following expression is obtained:

\[
\int_S \langle F(r)|\hat{n} \rangle \, dA = 3V \quad \text{(B.2)}
\]

For triangular meshes \( \hat{n} \) is constant for all points on a triangle. The point on which to evaluate \( F(r) = r \) can be chosen arbitrarily over the triangle since the scalar product of all vectors to points on a triangle with that triangle’s normal vector is constant. Therefore, the surface integral of \( F(r) \) over the surface becomes a summation of scalar products over all triangles.

The cross product between two vectors on the triangle can be used to calculate both the normal of the triangle (clockwise ordered triangles are used) and its area:

\[
(t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) = 2A_i \hat{n}_i \quad \text{(B.3)}
\]

Where \( t_{i,j} \) is the position of vertex \( j \) in triangle \( i \), \( A_i \) is the area of the triangle and \( \hat{n}_i \) is its normal.

Using this the surface integral becomes:
\[ \int_{S} (F(r)|\hat{n}) \, dA = \sum_{i} \left\langle \frac{t_{i,1} + t_{i,2} + t_{i,3}}{3} \mid (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \right\rangle \]

\[ = \frac{1}{6} \sum_{i} \langle t_{i,1} + t_{i,2} + t_{i,3} | (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \rangle \]

And thus the volume is given by:

\[ V = \frac{1}{18} \sum_{i} \langle t_{i,1} + t_{i,2} + t_{i,3} | (t_{i,3} - t_{i,1}) \times (t_{i,2} - t_{i,1}) \rangle \]

(B.4)
Appendix C

Momenta Conservation in Position Based Dynamics

Constraints that are invariant over rigid transformations conserve both linear and angular momenta when the system they constrain is moved in the direction of the gradient of the constraint function.

Denote a constraint function $C(X)$ and a rigid transformation $T(X)$. Then, for constraints that are invariant over rigid transformations:

$$C(X) = C[T(X)]$$

A rigid transformation in 3 dimensions has 6 degrees of freedom, and thus need 6 parameters to be fully determined:

$$T(X) \equiv T(p_1, p_2, p_3, p_4, p_5, p_6)(X)$$

This can be used in the earlier equality:

$$C(X) = C[T(X)] \iff \frac{d}{dp_i} C[T(X)] = 0, \forall i$$

Expanding this gives:

$$\frac{d}{dp_i} C[T(X)] = \left\langle \nabla_{T(X)} C[T(X)] \right| \frac{d}{dp_i} T(X) \right\rangle = 0, \forall i$$

Thus the gradient of the constraint function $\nabla_X C(X)$ is orthogonal to all the rigid body modes of the system meaning that moving the system in the direction of the gradient will not introduce any rigid movement and thus no momenta (given that all the points in the system have identical mass).
Appendix D

Simulator - Control Program Protocol

This is a description of the protocol used to communicate between the simulator and the control program in Cell-Lab.

The protocol is text based. Each command and its arguments are one line long. The commands can be separated in 3 different classes: Control commands (sent from CP to sim), reports (sent from sim to CP) and separation commands (of which there are 2: TICK(S) and DONE). The command names are always in capital letters and first on their line. Arguments are separated by whitespaces.

D.1 Separation commands

**TICK(S)**

Syntax: TICK(S n)

Sent from CP to simulator to initiate one or more time steps. If TICK is used, one time step will be performed. If TICKS n is used, <n> time steps will be performed. When the time step(s) is finished reports will be sent to CP followed by DONE.

Examples:

- **TICK**
- **TICKS 1**

These are equivalent.

**DONE**

Syntax: DONE

Sent from simulator to CP when time step(s) are done and all reports have been sent.
D.2 Control Commands

NEW_CELL

Syntax: `NEW_CELL position radius id color cadherin-type`

Creates a new cell in the simulation. Position and color are given as triplets of real numbers, the radius is one real number and id and cadherin-type are strings.

Example: `NEW_CELL 0 0 1 2 test_cell 0.5 0.5 0.9 none`

This command will create a cell at position (0,0,1) of radius 2 called test_cell colored in a blueish color without any cadherin.

SPLIT_CELL

Syntax: `SPLIT_CELL id direction`

Will initiate a split in the cell id. Id is as given to the cell in NEW_CELL or RENAME_CELL. The cell will be split in the plane given by direction. The two child cells will have the names `<id>a` and `<id>b`. Note that this call will only initiate a cell split, the split wont be finished for many time steps.

Example: `SPLIT_CELL test_cell 0 1 1`

Will split test_cell into test_cella and test_cellb. test_cella will be in the direction (0,1,1) of test_cellb.

KILL_CELL

Syntax: `KILL_CELL id`

Will remove the cell given by id from the simulation.

Example: `KILL_CELL test_cell`

test_cell will be removed from simulation.

RENAME_CELL

Syntax: `RENAME_CELL id new_id`

The cell `<id>` will henceforward be referred to as `<new_id>`. Useful to give child cells more descriptive names than the automatically generated.

Example: `RENAME_CELL test_cella muscle_cell`

test_cella will be renamed to muscle_cell.

COLOR_CELL

Syntax: `COLOR_CELL id color`
The cell <id> will be recolored in the graphical representation to <color>
Example: COLOR_CELL test_cell 1 1 1
Sets the color of test_cell to white.

### CADH_CELL

Syntax: CADH_CELL id cadh
Sets the **cadherin** type of the cell <id> to cadh.

### RESIZE_CELL

Syntax: RESIZE_CELL id new_radius
Sets the target radius of cell <id> to <new_radius>

### MOVE_CELL

Syntax: MOVE_CELL id position
Moves center of cell <id> to <position>.

### CELL_VELOCITY

Syntax: CELL_VELOCITY id velocity
Sets the average velocity of the cell <id>s vertices to <velocity>.

## D.3 Reports

All reports use a colon (‘:’) to separate between the report name and the data. All these reports will be sent from the simulator to CP when a time step(s) is done, followed by ’DONE’:

### CELL_LIST

Format: CELL_LIST: id_1, id_2, ..., id_n
Returns a list of all cell ids. The order of cells in this list is important since some other reports will reference cells by their index in this list rather than by id.

### CONTACT_LIST

Format: CONTACT_LIST: (id_1, id_4), (id_2, id_8), ...
Returns pairs of cells that are in contact, either by cell adhesion or other causes.


DISTANCE MATRIX

Format: DISTANCE MATRIX: d_{1,2}, d_{1,3}, \ldots, d_{1,n}; \ d_{2,3}, d_{2,4}, \ldots, d_{2,n}; \ldots; d_{n-1, n};

Returns a strictly upper triangular matrix of distances between cell centers. The order of the cells is the same as in CELL_LIST. \( d_{n,m} \) denotes the distance between cell \( n \) and cell \( m \). Columns are separated by commas and rows are separated by semicolons (as in matlab).

Example: DISTANCE MATRIX: 0.1, 0.2; 0.1;
A simple distance matrix between 3 cells.

CELL POSITIONS

Format: CELL POSITIONS: p_1, p_2, \ldots, p_n
The positions of the centers of all cells separated by commas. The order of the list is the same as in CELL_LIST.

CELL VELOCITIES

Format: CELL VELOCITIES: v_1, v_2, \ldots, v_n
The average velocity of all vertices in all cells separated by commas. The order of the list is the same as in CELL_LIST.

CELL VOLUMES

Format: CELL VOLUMES: V_1, V_2, \ldots, V_n
The volumes of all cells separated by commas. The order of the list is the same as in CELL_LIST.

TOTAL VOLUME

Format: TOTAL VOLUME: V
The total volume of all cells in the simulation. This should remain roughly the same throughout the simulation unless RESIZE_CELL, KILL_CELL or CREATE_CELL is used.

SPLITTING LIST

Format: SPLITTING LIST: id_1, id_4, \ldots
Returns the names of all cells that are currently splitting in the simulation.
Appendix E

Configuration File Format

The configuration file is a text file with settings that don’t change during the simulation run time. The file is divided into different sections to promote proper structuring of the file.

Section delimiters are enclosed in square brackets. Settings are on the format: 

`SETTING_NAME=VALUE`. There can be only one section delimiter or setting per line. Lines that start with a hash sign (`#`) are not read.

There are 3 different sections: cell-params, world-params and outputs.

E.1 Cell Parameters

The settings in the ‘cell-params’ section determine properties for the cells in the simulation including what cell model to use and specific parameters on the chosen model.

model

Determines the cell model to use. Can be either `pairs` for pair-pressure model, `fbsm` for full body shape matching model or `ssm` for surface shape matching model. These are described in detail in the report.

lod

Cell level of detail. 0 is the lowest level of detail. Must be an integer number. The number of vertices on each cell increases roughly with the square of this number.

density

The cell’s density in mass per volume units. This determines the cell’s total mass. A real number.
ssm-theta-sigma

The cutoff radius for a patch in radians. A positive real number.

ssm-stiffness

The stiffness value used when relaxing ssm constraints. A real number in the range $[0, 1]$.

ssm-fbsm-stiffness

The stiffness of the fbsm constraint if used together with ssm. Using fbsm constraints together with ssm helps preventing cells from reaching highly deformed states.

fbsm-stiffness

Stiffness used when relaxing fbsm constraints.

pairs-stiffness

Stiffness used when relaxing pair constraints.

volume-stiffness

Stiffness used when relaxing volume constraints.

cell-separation-stiffness

Stiffness of cell-cell separation constraints.

cell-cell-friction-coeff

The amount of friction between cells. A real number in the interval $[0, 1]$ where 1 means no friction and 0 total friction.

E.2 World Parameters

This “world-params” section sets parameters more general than those in “cell-params”.

dt

The simulation delta time. Less important than in strictly physical simulators since the result of a PBD simulation is more dependent on the number of time steps taken than on the time each step represent.
relaxation-steps

The number of solver iterations each time step.

damping-coeff

Viscous damping coefficient. A number between 0 and 1 that is multiplied into the velocities of the simulator every time step.

shell-type

The type of egg shell that surrounds the cells. Can be either hard-shell for a hard spherical shell or free-space for no egg shell at all.

shell-radius

The radius of the surrounding egg shell if a spherical shell is used. If an elliptic shell is used, this radius determines the volume of the ellipsoid. A real positive value.

shell-eccentricity

The eccentricity of the shell. If left out, or set to zero, the shell will be spherical.

shell-friction-coeff

The amount of friction between a cell and the shell, if any. A number between 0 and 1 that is multiplied into the velocities of the simulator every time step.

E.3 Output Parameters

These settings control different outputs of the simulator. Note: this has nothing to do with the reports sent from the simulator to the controller program (CP).

Available outputs are: geometry dumps and view dumps. Geometry dumps save all the positions of the simulator to a file that later can be replayed to watch the development of the simulation off line. View dumps are simply screen shots taken at every time step. These can for example be used to create videos of the simulation.

enable-geometry-dump

This enables or disables geometry dumps of the simulation. Must be either 0 or 1.
geometry-dump-file

The geometry dump file.

enable-view-dump

Enables or disables view dumps.

view-dump-folder

The folder the screen shots will be in.

view-dump-prefix/numbers

These two settings determine the name of the view dump files. View-dump-prefix is a name that all files will begin with, and view-dump-numbers determine the number of numbers used to identify individual files. If prefix is vdump and numbers is 5 the first file will be: "vdump00001.png". Using 5 numbers to identify the images limits the number of saved images to 99999.

E.4 Skeleton File

Here is a skeleton configuration file:

```plaintext
[ cell−params ]

# model can be pairs , ssm (surface shape matching)
# or fbsm (full body shape matching)
model=ssm
# level of detail of cells (~ number of vertices)
lod=5
# mass units per unit volume
density=1
# The falloff of ssm weighing function
ssm−sigma=0.15

[ world−params ]

# Timestep
dt=0.05
# Global viscous damping
damping_coeff=0.94
# Can be hard_shell or free_space
```
shell − type=hard _shell
shell − radius=5
# Global number of relaxation steps
relaxation − steps=1

[outputs]

enable − geometry − dump=0
geometry − dump − file=geometry .dump
enable − view − dump=0
view − dump − folder=view − dumps
view − dump − prefix=vdump
view − dump − numbers=5
Appendix F

Using The Simulator

The simulator takes 3 command line arguments: -c, -i and -o. -c gives the name of the configuration file to use, -i and -o gives the name of the files to use as input and output pipes. Passing a hyphen to -i or -o tells the simulator to use standard input and output as these pipes.

Example:

> cell-lab -c conf.file -i to_sim_pipe -o from_sim_pipe

In this example conf.file is the file described in E. to_sim_pipe/from_sim_pipe are named pipes that the controller program (CP) writes and reads to and from. On posix systems named pipes can be created with the command `mkfifo`:

> mkfifo to_sim_pipe

For testing purposes, the built in command `cat` can be used as a very simple CP:

> cat | ./cell_lab -c conf.file -i - -o -

Now the user can type commands into the console, cat will print them to its standard output which is piped to Cell-Lab’s standard input. The -i - tells cell-lab to use standard input as control channel, so the simulation can be controlled directly by the user. Output from the simulator’s standard out (which includes the reports) and error will be printed in the console.

This is a tiresome way to use cell-lab, but gives some insight to how the controller program should be written.

Note: the standard out is used for some diagnostics output, even when simulation reports are sent there. To make life easier for the CP named pipes are encouraged.
Note 2: many high level languages use buffering when reading and writing to files. This interferes with the communication between the CP and the simulator since control commands and reports can be delayed for a long time before the buffer is flushed and anything is actually written to file causing unexpected freezes of the program. Buffering thus must be disabled for the communication pipes. In perl, which the example controller program is written in, this is done by doing:

```perl
my $to_sim_file;
open ($to_sim_file, '>', 'to_sim_pipe') or die ('could not open to_sim_pipe');
my $old_sel = select $to_sim_file;
$| = 1;
select $old_sel;
```

$| in perl is a special variable that controls whether or not the currently selected file handle should operate in unbuffered or buffered mode. Consult your language reference to determine how this is done in the language of your choice.
Bibliography


