

Cell-based computational modeling of vascular morphogenesis using Tissue Simulation Toolkit v2.0

Josephine T. Daub¹ and Roeland M.H. Merks^{2,3*}

¹Independent researcher.

^{2*}Mathematical Institute, Leiden University, Einsteinweg 55, Leiden, 2333 CC, The Netherlands.

³Institute of Biology Leiden, Leiden University, Einsteinweg 55, Leiden, 2333 CC, The Netherlands.

*Corresponding author(s). E-mail(s): merkstrmh@math.leidenuniv.nl;

Abstract

Computational modeling has become a widely used tool for unraveling the mechanisms of higher-level cooperative cell behavior during vascular morphogenesis. However, experimenting with published simulation models or adding new assumptions to those models can be daunting for novice and even for experienced computational scientists. Here, we present a step-by-step, practical tutorial for building cell-based simulations of vascular morphogenesis using the Tissue Simulation Toolkit (TST). The TST is a freely available, open-source C++ library for developing simulations with the two-dimensional Cellular Potts model, a stochastic, agent-based framework to simulate collective cell behavior. We will show the basic use of the TST to simulate and experiment with published simulations of vascular network formation. Then, we will present step-by-step instructions and explanations for building a published simulation model of extracellular-matrix-guided angiogenesis. Demonstrated mechanisms include cell-cell adhesion, chemotaxis, cell elongation, haptotaxis, and haptokinesis.

Keywords: Cellular Potts Model, agent-based modeling, Tissue Simulation Toolkit, angiogenesis, cell-based model, parameter study, quantification, Glazier-Graner-Hogeweg model

Revision process and responsible use of AI

This revision, prepared for the third edition of the book and representing the second version of this chapter, was developed by the authors with assistance from OpenAI’s ChatGPT. The authors remained responsible for the scientific content, modelling assumptions, software-design decisions, pedagogical aims, and final wording. AI assistance was used for technical and editorial tasks, including revising text, identifying inconsistencies, checking code and references, and proposing software changes and tests. All AI-generated suggestions were reviewed by the authors before inclusion.

The revision proceeded iteratively in five passes. In the *first pass*, the scope was defined and the original chapter was migrated to the current Springer Nature L^AT_EX format, updating terminology, installation instructions, links, and references. The *second pass* updated the tutorial from the older TST version to TST 2.0, preserving the scientific model while replacing obsolete implementation details. The *third pass* focused on scientific and pedagogical clarity and consistency between text, equations, code, parameters, figures, and literature. The *fourth pass* concentrated on technical correctness and backward compatibility: the AI assisted in tracing software changes and proposing patches and tests, while the authors made the design decisions and compiled, ran, and inspected the software. The *fifth pass* resolved remaining author comments and systematically checked the manuscript, code fragments, bibliography, cross-references, and L^AT_EX compilation. Thus, AI-assisted review complemented, but did not replace, human scientific judgement and software validation.

Estimated energy use

We also attempted to account for the energy used by this AI-assisted revision. ChatGPT does not provide per-conversation energy telemetry, so this cannot be measured directly. Our revision involved an order of magnitude of 10^2 substantive AI interactions, many with long manuscript or source-code contexts. Oviedo et al. estimate a median electricity use of 0.31 Wh per standard frontier-scale AI query (interquartile range 0.16–0.60 Wh) [1], consistent with an independent production-scale measurement of 0.24 Wh for a median Gemini Apps text prompt [2]. Oviedo et al. also estimate that long reasoning queries can require more than an order of magnitude more energy, with a 13-fold increase in one representative scenario.

For roughly 100 interactions, these figures imply about 0.03 kWh if all resembled standard queries and about 0.4 kWh if all resembled the long-reasoning scenario. Given the mixture of interactions in this revision, we estimate an order of 0.03–0.5 kWh of AI inference electricity. This is an order-of-magnitude estimate, not a measurement, and excludes model training, end-user devices, and embodied infrastructure. We do not convert it to a carbon footprint because the serving locations and electricity mix are unknown.

1 Introduction

To understand vascular morphogenesis, we need to connect molecular and cellular mechanisms to collective behaviours such as sprouting, branching, and vascular network formation. Computational models provide a useful bridge between these scales because they force hypotheses about cell behaviour and cell–environment interactions to be stated explicitly and make their consequences testable. In a productive modelling cycle, however, reproducing a visually plausible pattern is only a first step. Different mechanisms can produce similar-looking networks, so models gain explanatory value when their dynamics and responses to perturbations are compared quantitatively with experiments. Discrepancies can then be used to reject, refine, or extend the hypotheses encoded in the model. This iterative view is central to systems biology [3] and is illustrated by our recent quantitative model-falsification work on endothelial network formation [4]. A complementary experimental and quantitative workflow for time-lapse imaging of endothelial network formation is described by Vergroesen and Merks elsewhere in this volume.

The formation of vascular networks *in vitro* is a well-studied example. When plated in Matrigel or other culture gels, endothelial cells, including human umbilical-vein endothelial cells [5] and bovine aortic endothelial cells [6], can organize into vascular-like network patterns. Mathematical and computational models have proposed several mechanisms for this collective behaviour. In mechanochemical continuum models, cells deform the surrounding matrix, generating mechanical and density cues that can guide neighbouring cells by haptotaxis [7–9]. Other continuum models proposed attraction through self-generated chemical gradients [10–12], related to the chemotactic aggregation mechanisms studied in *Dictyostelium* [13]. These models describe cells through continuous density fields and are therefore particularly suitable for analysing tissue-scale pattern formation.

Cell-based models instead retain individual cells and thus allow hypotheses about cell shape, cell–cell contacts, and cell–matrix interactions to be expressed directly [14]. Cellular Potts models (CPMs) [15], for example, have shown how cell elongation can shift collective behaviour from compact aggregates toward network-like patterns [16, 17], how contact-inhibited chemotaxis can organize endothelial cells [18], and how preferential interactions with elongated neighbours can promote multicellular sprouting [19]. CPMs and related cell-based models can also represent interactions with extracellular matrix and matrix-bound signals [20–22], couple cell behaviour to subcellular signalling [23], and address lumen formation [24]. They have been embedded in larger models of tumour growth [25], neovascularization in the eye [26], and multiscale vascular development and intervention [23, 27]. Recent developments have made cell–ECM interactions more explicit by coupling cell-based models to deformable fibrous matrix representations. In a recent model of zebrafish intersegmental vessel formation, endothelial self-organization was coupled to a deformable ECM network; the model predicted that reduced matrix density slows sprouting and increases ectopic vessel fusion, predictions that were subsequently tested experimentally [28]. The continuum ECM representation developed in this chapter is therefore one member of a broader family of approaches for investigating how extracellular matrix properties guide vascular morphogenesis. The field has consequently moved from asking whether

a proposed mechanism *can* generate a vascular pattern toward asking which mechanisms remain consistent with quantitative, time-resolved data. A recent comparison of several computational models with endothelial network-formation experiments showed that models producing superficially similar networks can differ substantially in their remodelling dynamics; in those experiments, a cell-elongation model reproduced key dynamical features most closely [4]. For sprouting angiogenesis, classical “snail-trail” models represented sprout extension primarily through migration of a leading tip, with the stalk forming behind it; an influential example is the continuous/discrete model of Anderson and Chaplain [29]. Experimental observations of endothelial rearrangements and cell mixing have motivated models that move beyond this picture, as reviewed by Stepanova *et al.* [30]. These developments reinforce a useful modelling principle for this chapter: implement biological assumptions in a form that makes them easy to vary, then compare the resulting behaviour with appropriate experimental observables rather than treating visual agreement as validation.

Reviews of vascular-network modelling and cell-based vascular morphogenesis can be found in [31, 32], with broader perspectives on vascular development in [33] and on cell rearrangements during sprouting in [30]. Here we take a practical approach. The goal is to show how a biological hypothesis can be translated into a working cell-based model, how individual mechanisms can be introduced one at a time, and how the model can then be used for computational experiments. The worked example is the extracellular-matrix-guided angiogenesis model of Daub and Merks [22].

Several open-source environments support cell-based and Cellular Potts modelling. CompuCell3D [34] provides a high-level environment for constructing two- and three-dimensional multicellular models and coupling CPMs to additional model components. Morpheus [35] provides a declarative, XML-based environment that combines multicellular models with reaction–diffusion and ODE models. Other open-source cell-based frameworks include Chaste [36], and FLAME [37]. These packages differ in modelling formalism, level of abstraction, and intended workflow.

The Tissue Simulation Toolkit (TST) occupies a deliberately source-oriented niche. It is intended for researchers who want to understand, modify, and extend CPM algorithms and who want the biological assumptions of a model to remain visible in comparatively small pieces of C++ code. This makes TST particularly useful for a tutorial in which the purpose is not only to run a published model, but also to see precisely where each modelling assumption enters the implementation.

Design principle: keep the framework generic, and write the biological hypotheses in a separate model file.

TST 2.0 makes this principle more explicit than earlier versions of the Tissue Simulation Toolkit. Generic operations—such as performing a Cellular Potts copy attempt, integrating reaction–diffusion equations, storing cells, and rendering simulation output—belong to the TST framework. The biological hypothesis being tested—for example, which substances a cell secretes, how an extracellular matrix is remodelled, or when a cell divides—should, whenever possible, be expressed in the model code. Throughout this chapter we therefore build the angiogenesis model progressively, adding one biological assumption at a time while leaving the

generic simulation machinery unchanged unless a genuinely new generic mechanism is required.

This separation is also a practical guide for extending TST: before changing the framework, first ask whether the new assumption can instead be expressed in model code, with its adjustable choices exposed as parameters. In TST 2.0 the answer is more often “yes” than it was in the version used for the first edition of this chapter.

2 Materials

The following materials and prior knowledge are required for using and extending the code provided in this chapter.

2.1 Required software: C++ compiler and libraries

Using the TST requires basic knowledge of the use of the terminal and a C++ compiler. To acquire these skills, please refer to the appropriate books or internet tutorials.

Windows: The easiest way to install and work with TST 2.0 on Windows is through the Windows Subsystem for Linux (WSL2), using an Ubuntu installation. Install WSL2 and Ubuntu following Microsoft’s current WSL installation instructions (<https://learn.microsoft.com/windows/wsl/install>). Then open the Ubuntu terminal and first update the package index:

```
sudo apt update
```

If Ubuntu cannot locate one of the Qt development packages used below, enable the Ubuntu **universe** repository and update the package index again:

```
sudo add-apt-repository universe
sudo apt update
```

After this initial WSL setup, follow the Linux installation and command-line instructions below. The chapter assumes only basic terminal use: changing directories, copying files, and running the commands shown explicitly.

macOS: Install the Xcode development environment from Apple, including the Command Line Tools. We recommend installing Homebrew and using it to install the additional libraries required by TST:

```
brew install qt libpng zlib
```

Linux: TST requires C and C++ compilers, the standard binary utilities and **make**, as well as Git, zlib, libpng, OpenCL, and Qt 6. On recent Ubuntu and other Debian-based systems, first update the package index and then install the dependencies:

```
sudo apt update
sudo apt install gcc g++ binutils make git zlib1g-dev \
  libpng-dev ocl-icd-opencl-dev qt6-base-dev
```

If asked to continue, type Y. Other Linux distributions provide equivalent packages under similar names.

Qt note: TST 2.0 uses Qt 6 on both macOS and Linux. The current source has been tested with Homebrew Qt 6.11 on macOS; on Ubuntu, `qt6-base-dev` provides the required development tools and libraries. If a suitable Qt 6 installation is already present, do not install a second copy. Instead, ensure that its `qmake` is available; depending on the installation, it may be necessary to adjust `src/Tissue.Simulation.Toolkit.pri` so that `qmake` can find Qt, libpng, and zlib.

2.2 Source code of the Tissue Simulation Toolkit

Get the source code of the Tissue Simulation Toolkit from <https://github.com/mathbioleiden/Tissue-Simulation-Toolkit>. This tutorial is based on TST version 2.0.

The preferred way to obtain the source is to use the version control system `git`. From your home directory (`cd ~`), clone the TST 2.0 branch using the HTTPS address below. For a public repository, HTTPS normally allows cloning without a GitHub account or SSH key. If your local Git configuration redirects GitHub URLs to SSH, however, Git may still request SSH credentials. Clone the branch, including its submodules, by typing:

```
git clone --recursive -b TST2.0 \
  https://github.com/mathbioleiden/Tissue-Simulation-Toolkit.git
```

The option `--recursive` initializes the Git submodules required by the complete TST source tree. Some optional submodules are not used by this chapter, but cloning recursively gives a self-contained TST 2.0 checkout and avoids missing-dependency errors during the standard build.

Next, move into the TST folder:

```
cd Tissue-Simulation-Toolkit
```

and create and switch to a new branch for the changes made while working through this chapter:

```
git checkout -b mybranch
```

where `mybranch` can be replaced by a name of your choice. We encourage you to use Git to keep track of your changes as you proceed through this book chapter.

If you cloned the repository without the `--recursive` option, obtain the missing submodules afterwards by typing, from the top-level TST folder:

```
git submodule init
```

```
git submodule update
```

If you prefer not to use Git, the source can also be downloaded as an archive from GitHub. However, GitHub source archives do not automatically include Git submodules. For that reason, cloning the repository recursively is strongly recommended for this chapter.

2.3 Compile the Tissue Simulation Toolkit

Open a terminal and go to the top-level directory in which TST was installed, e.g.,

```
cd ~/Tissue-Simulation-Toolkit
```

Compile the standard C++ TST models by typing:

```
make
```

If `make` reports that `qmake` cannot be found, see [Note 1](#). The top-level `Makefile` runs the required `qmake` and `make` steps for the individual models and places the resulting executables in the `bin/` directory. The simulations in this chapter use only these standard C++ components. In particular, the optional `with_adhesions` build target and its additional Python, MUSCLE3, and molecular-dynamics dependencies are not required for this chapter.

The first complete build may take some time. Later in the chapter, after creating the new `ecm` model, you can compile only that model with `make bin/ecm` from the top-level TST directory.

If compilation reports that `lib/MultiCellDS/v1.0/v1.0.0/libMCDS/xsde` cannot be found, the Git submodules were probably not initialized. Return to the top-level TST folder, run `git submodule init` followed by `git submodule update`, and then repeat `make`.

2.4 Test the Tissue Simulation Toolkit

If compilation was successful, the `bin/` folder contains an executable called `vessel`. Change to that directory:

```
cd bin
```

and run the example chemotaxis simulation using the supplied parameter file:

```
mkdir -p ../out
```

```
./vessel ../data/chemotaxis.par
```

The `out/` directory is used for image output; creating it beforehand prevents PNG-write errors in versions of TST that do not create it automatically. When using WSL2 on Windows, use the same commands in the Linux terminal. If all goes well, a window

appears with output similar to Figure 1. You can stop a running simulation by closing its graphics window or by pressing **Ctrl+C** in the terminal from which it was started.

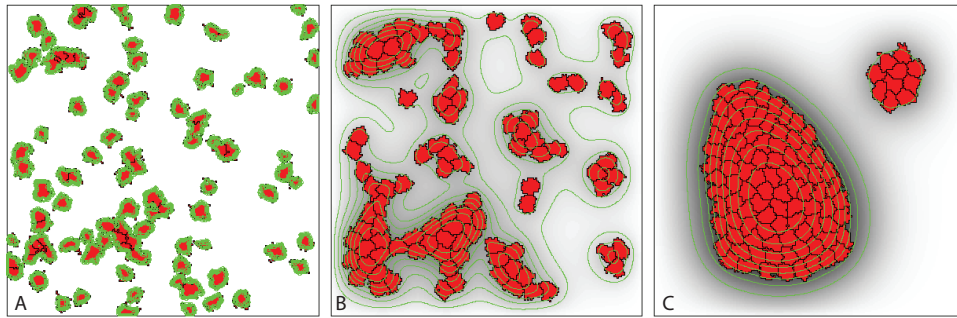


Fig. 1 Typical output of TST-model `vessel` with parameter file `chemotaxis.par`. The model is a Cellular Potts implementation of the Keller-Segel model [54] of cellular aggregation due to chemotaxis. All cells secrete a chemoattractant, which diffuses and is degraded in the surrounding matrix. (A) Initial condition; (B) configuration after 500 Monte Carlo Steps; (C) configuration after 10000 Monte Carlo Steps. Grey levels indicate chemoattractant concentrations (white: low concentration; black: high concentration); contour lines connect equal chemoattractant concentrations; cells shown as red patches delineated with black contours.

2.5 How TST 2.0 is organized

Before modifying a model, it is useful to distinguish three parts of a TST simulation.

Parameter definition. Parameters that can be changed without recompiling the program are declared centrally in `src/parameters/parameters.hpp` and are read from a `.par` file. In this chapter we use parameters whenever a quantity represents a model choice that the user may want to vary.

Model code. A model source file in `src/models/` contains the biological rules and the model-specific orchestration of the simulation. The supplied `vessel.cpp`, for example, defines the initial condition and time-step loop and supplies model-specific implementations for initialization and reaction terms of the partial-differential-equation (PDE) fields, which represent spatially varying quantities such as diffusible signals or extracellular matrix density. New models in this chapter will be constructed by starting from such a model file and extending it.

Framework code. The directories `src/cellular_potts/` and `src/reaction_diffusion/` contain generic CPM and PDE machinery. These classes implement operations that can be reused by many biological models. We modify this layer only when the biological mechanism cannot cleanly be expressed through an existing TST interface. Such a modification should be understood as extending the Toolkit itself, rather than merely defining a new model.

Conceptually, the relation between these parts is summarized in Figure 2.

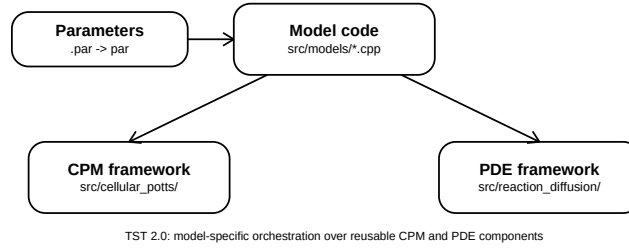


Fig. 2 Conceptual organization used in this chapter. Model code orchestrates the Cellular Potts model (CPM) and partial-differential-equation (PDE) framework components, while parameters from the `.par` file are available through the global `par` object.

Parameters from the `.par` file are read into the global `par` object, which is available to the model code and to framework components. The model, in turn, orchestrates the CPM and PDE components and supplies model-specific hooks where TST provides them.

This architecture determines how code modifications are presented below. Each programming step is identified as a **Parameter definition**, **Model code**, or **Framework code**. Most of the tutorial should take place in the first two categories. A framework modification is introduced only when it represents a mechanism that is potentially reusable beyond the particular angiogenesis model.

3 Methods

The *Tissue Simulation Toolkit* provides generic computational machinery for two-dimensional Cellular Potts models (CPMs), numerical reaction–diffusion models, and coupling between cells and continuous fields. A TST simulation combines this reusable framework with a comparatively small model program that states the biological assumptions. The supplied `vessel` model is an example: its model file specifies the initial endothelial cells, model-specific PDE dynamics and the sequence of operations performed during a time step, while the generic CPM and PDE algorithms are provided by the TST.

The exercises below follow the same separation. We first run and modify existing models through their parameter files. We then inspect how model code uses the TST framework. Finally, we construct the tumor-angiogenesis example incrementally. At each stage we first formulate the biological assumption and then decide whether it belongs in a parameter, in model code, or—only when necessary—in reusable framework code.

3.1 The Cellular Potts model: mathematical background

Before working through the TST examples, we briefly summarize the mathematical model implemented by the framework. The Cellular Potts model (CPM) represents biological cells on a regular lattice (a rectangular lattice in the TST). The lattice sites $\vec{x}$, contain integer values - called *cell ID* or *spins* - $\sigma(\vec{x}) \in \mathbb{Z}^{0,+}$, where each value of

σ identifies an individual biological cell or one of the surrounding extracellular matrix materials or interstitial fluids. A (usually connected) cluster of lattice sites $\vec{x}$ containing the same cell ID $\sigma(\vec{x})$ then represents a biological cell. The CPM represents amoeboid cell motility by simulating random pseudopod extensions and retractions. To this end, the algorithm iteratively picks a random lattice site $\vec{x}$ and a random adjacent site $\vec{x}'$ (*i.e.*, it selects a random adjacent lattice site pair $(\vec{x}, \vec{x}')$) and attempts to copy the spin $\sigma(\vec{x}')$ into lattice site $\vec{x}$, as if the cell $\sigma(\vec{x}')$ ‘overgrows’ cell $\sigma(\vec{x})$. Whether the copy attempt succeeds or not will depend on the active force the cell generates and the resulting reactive forces of the adjacent cells and materials.

These forces are proportional to the change in free energy H of the system due to the copy attempt, as $F \sim \Delta H$. The copy attempts are accepted with probability $P(\Delta H) = 1$ if $\Delta H < -H_0$ (in which case the cell attempted to extend a pseudopod in the direction of least resistance), and $P(\Delta H) = \exp\left(\frac{-\Delta H + H_0}{T}\right)$ if $\Delta H \geq -H_0$, representing active, random cell motility, with T a cell motility parameter, setting the probability by which the cell will make an energetically unfavorable move, and H_0 a “dissipation energy” parameter, setting the minimal energy that must be ‘overcome’ in order to make a move. Typically, $H_0 = 0$; a use for this parameter will be described in more detail below.

A typical form of a Hamiltonian describing differential cell adhesion is,

$$H = \sum_{(\vec{x}, \vec{x}')} J(\tau(\sigma(\vec{x})), \tau(\sigma(\vec{x}'))) [1 - \delta_{\sigma(\vec{x}), \sigma(\vec{x}')}] + \lambda_A \sum_{\sigma} (A(\sigma) - a(\sigma))^2, \quad (1)$$

where a copy attempt leading to a drop in the Hamiltonian will be accepted with high probability. The first term in the Hamiltonian describes cell-cell and cell-ECM adhesion, and is a sum over all adjacent lattice pairs $(\vec{x}, \vec{x}')$ with the Kronecker delta term selecting only interfaces between unlike cell IDs ($\delta_{ij} = 1$ if $i = j$ and 0 otherwise). The cell-cell interfaces are associated with a positive binding energy $J(\tau_1, \tau_2)$, where high values of $J(\tau_1, \tau_2)$ give weak adhesion and lower values give stronger adhesion. To keep the number of values for J limited, they are defined between *cell types* τ , where each cell ID σ is associated with one of a small number of cell types, $\tau(\sigma)$. The second term constrains the area that the cells cover, with $A(\sigma)$ a resting area, and $a(\sigma)$ the actual, potentially compressed or expanded area of the cells.

The Hamiltonian can be extended in many ways, to have the Cellular Potts model describe additional cell behaviors, such as chemotaxis, cell shape constraints, and so forth. The remainder of this chapter will describe in detail how to do so using the TST. Section 3.2 describes the basic usage of the TST, demonstrating how to run a simulation of cell sorting [15, 38] and published simulations of angiogenesis and vasculogenesis [16, 18]. Sections 3.3 and 3.4 provide implementation details of the CPM. The main portion of this chapter is Section 3.5, which demonstrates how to extend the TST and provides stepwise implementation instructions for the ECM-guided angiogenesis model of Daub and Merks [22].

3.2 General usage of Tissue Simulation Toolkit

Exploring the sorting model

We will start by exploring a model of differential-adhesion-driven cell sorting, the tendency of initially mixed cell populations with different adhesive properties to rearrange into reproducible spatial organizations, such as segregation or engulfment. This model provides a simple illustration of how local differences in interfacial energy can generate tissue-scale organization [15, 39].

We begin without changing any C++ code. A compiled TST model is an executable program, while the corresponding `.par` file specifies parameter values for a particular simulation. Thus, the first level at which a user experiments with a model is the parameter file: changing a parameter changes the biological or numerical assumptions of a run without changing the program itself.

The main simulation loop is defined in a model source file in `src/models/`, with a corresponding executable in `bin/`. TST 2.0 includes, among others, the example models `sorting.cpp` and `vessel.cpp`. We will illustrate the use of the Tissue Simulation Toolkit with a simulation of differential-adhesion driven cell sorting [15, 38] according to the Hamiltonian defined above with two cell types.

1. The cell sorting simulation is defined in `src/models/sorting.cpp`. The standard TST 2.0 build compiles this model together with the other C++ examples, so no project file needs to be edited. If necessary, return to the top-level TST folder and type `make`.
2. Change to the `bin/` directory, if you are not already there.
3. Start the program by typing the executable name, followed by the path to the parameter file:

```
./sorting ../data/sorting.par
```

A window will appear with the initial cluster, containing a mixed pattern of cells.

4. Now let's try to change the parameters in order to get the cells to stick together and sort out, or to make one type of cells to engulf the other type. To do so, first make a copy of the file `J.dat`, by typing:

```
cp ../data/Jnoadhesion.dat ../data/myJ.dat
```

On Linux, macOS, and Windows/WSL2 the same command can be used from the `bin/` directory. Open `data/myJ.dat` in a text editor. It will look something like this:

```
3
0
20 40
20 40 40
```

The “J-file” describes the adhesion energies as a diagonally symmetric matrix J ; the first line gives the number of cell types, 3. That is two cell types (red, yellow) plus one for the ECM (white).

The second line gives $J(M, M)$, the adhesion energy between “ECM cells”. Since we only have one ECM “cell”, there are not boundaries between cells of type “ECM”; thus $J(M, M) = 0$ by definition. The next line describes the adhesion energies between cell type 1 and the other cell types and the ECM. The first number is the adhesion energy between cell 1 and the ECM, $J(M, 1) = J(1, M)$, the second number the adhesion energy, $J(1, 1)$ of cell 1 to its own type. Similarly, the fourth line lists $J(M, 2)$, $J(1, 2)$, and $J(2, 2)$. Here M denotes the extracellular space (the CPM state $\sigma = 0$), whereas 1 and 2 denote the two cell types.

5. To make the TST look for your new “J-file”, open `data/sorting.par` in a text editor and change its `Jtable` entry so that it points to the copied file. In the TST 2.0 version used for this chapter, input-resource paths are resolved relative to the parameter file. Because `myJ.dat` and `sorting.par` are both in `data/`, set:

```
Jtable = myJ.dat
```

See also Step 7 for more information on the format of the parameter files.

6. Experiment with different values of J . For example, to have the cells of type 1 stick together, set the parameters such that $J(1, 1) < 2J(M, 1)$. In this way a boundary between two cells of type 1 is energetically more favorable than two cell-ECM boundaries of the same length. The factor of two arises because separating two contacting cells by matrix replaces one cell-cell interface by two cell-matrix interfaces. We leave the other cases as an exercise to the reader.
7. All the other parameters are defined in the parameter file with extension ‘.par’. The format of this file is as follows. All lines starting with ‘#’ are treated as comments and will be ignored. Parameter values are assigned by giving the parameter name, a mandatory space, then the ‘=’ sign, followed by another space, and the parameter value. For example, the following line sets the value of λ in the Hamiltonian:

```
lambda = 50
```

Similarly, `T` defines the value of the cell motility parameter, and `target_area` gives the value of the target area, A . The parameter file also gives the values of additional parameters not mentioned in the CPM model definition above; an overview is given in Table 1.

Exploring the vessel model

We will next describe the basic structure and usage of two previously published models of angiogenesis and vasculogenesis [16, 18]. Both of these models assume that endothelial cells secrete a diffusive chemoattractant (e.g. VEGF), which is simulated using a continuum, partial-differential equation model. A PDE model is coupled to the CPM

to describe the chemoattractant, c ; it is secreted by the endothelial cells, it diffuses, and it is degraded in the extracellular matrix,

$$\frac{\partial c}{\partial t} = D\nabla^2 c + \alpha(1 - \delta(\sigma(\vec{x}), 0)) - \epsilon\delta(\sigma(\vec{x}), 0)c \quad (2)$$

where D is the diffusion coefficient, α the secretion rate, and ϵ the decay rate. The indicator $\delta(\sigma(\vec{x}), 0)$ equals 1 in the medium and 0 at lattice sites covered by cells. We implement chemotaxis by assuming that cells preferentially extend pseudopods in the direction of chemoattractant gradients [40]. The algorithm allows for an extra energy drop at the time of copying:

$$\Delta H_{\text{chemotaxis}} = -\mu(c(\vec{x}') - c(\vec{x})) \quad (3)$$

where $(\vec{x}, \vec{x}')$ is an adjacent lattice-site pair and μ is the chemotactic strength.

8. The program `src/models/vessel.cpp` implements the angiogenesis and vasculogenesis simulations. If all went well, you compiled this program in Section 2 and the executable `vessel` is in the `bin/` directory. If it is missing, return to the top-level TST directory and type `make`.
9. Start up a Cellular Potts simulation with the parameter file `chemotaxis.par`, by typing on the command line:

```
./vessel ../data/chemotaxis.par
```

10. Reduce the display frequency to observe the long time behavior of the simulation. Edit the parameter file, setting `storage_stride = 100`.

As in the PDE model by Gamba *et al.* [10], the cells secrete a chemoattractant (shown in grey and with the green concentration isolines), and move towards higher concentrations of the attractant. The endothelial cells should form rounded aggregates as in Figure 1.

11. Before experimenting with the parameters, make a copy of the parameter file:

```
cp ../data/chemotaxis.par ../data/mychemotaxis.par
```

Experiment with the parameters, including the decay rate of the chemoattractant (`decay_rate`), the diffusion coefficient (`diff_coeff`), and the number and size of the cells (`n_init_cells`, `target_area`). Stop the simulation by pressing `Ctrl+C` in the Terminal or closing the graphics window. Edit `mychemotaxis.par` as described in Step 5. Then restart the simulation with `./vessel ../data/mychemotaxis.par`. A recommended exercise is to study what determines the size of the cell clusters. See also Note 2 for numerical instability issues. With additional assumptions, including elongated cells [16] or contact inhibition [18] this chemotactic mechanism suffices for producing vascular-like

structures. To mimic cell elongation due to cytoskeletal remodeling we add a cell length constraint to the Hamiltonian:

$$H' = H + \lambda_L \sum_{\sigma} (l_{\sigma} - L_{\sigma})^2, \quad (4)$$

where l_{σ} is the length of cell along its longest axis, L_{σ} its target length, and λ_L is the strength of the length constraint. Assuming that cells are ellipses, we can derive their length from the largest eigenvalue of their inertia tensor. The length constraint could cause cells to split into disconnected patches. We prevent this artifact by introducing a connectivity constraint, which reflects the physical continuity and cohesion of the actual cell. This connectivity constraint sets the “dissipation energy” parameter, H_0 , to a large positive value for copy updates that would locally break the connectivity of the cell (see [16] for detail).

12. Start from a clean parameter file, by copying `chemotaxis.par` and editing the copy. Experiment with the value of `target_length` (L_{σ}); a good value to start with would be `target_length = 60` ($L = 120 \mu\text{m}$, if `dx=2.0e-6`).
13. Experiment with small numbers of cells, say `n_init_cells = 10`. What happens to the polygonal pattern?
14. Increase the field size using the parameters `size_x` and `size_y`, *e.g.*, set both values to 500. A useful parameter to change is also `subfield`; if you set this to `subfield = 1.5` the cells will be distributed within a restricted space in the center of the field so the network has sufficient space to expand. Note that increasing the field size also increases the computational cost and therefore slows down the simulation.

An alternative mechanism for network formation and sprouting is contact inhibition of chemotaxis. In this mechanism, we assume that only the cell-ECM interfaces sense the chemoattractant, a mechanism that phosphorylation of VEGFR-2 receptors by vascular-endothelial cadherin might mediate (see [18] for detail).

15. Start with a clean copy of `chemotaxis.par`. Make sure that the cell shape is constrained (`lambda2 = 5.0`). Switch on the contact-inhibition mechanism: `vecadherinknockout = false`. Save the parameter file, and restart the simulation. If all proceeds well, the cells will form more elongated aggregates.
16. To see a network of cells, increase the field size and increase the number of cells accordingly. Suitable values are: `size_x = 500`, `size_y = 500`, `n_init_cells = 500`, `subfield = 1.5`. To see a network develop more quickly, within the standard field of 200×200 , reduce the diffusion length (width of the chemoattractant gradients), *e.g.*, by increasing the secretion and degradation rates of the chemoattractant, `decay_rate = 1e-3`, `secre_rate = 1e-3`, the values used in [18], and use a larger number of smaller cells, `target_area = 50`, and `n_init_cells = 300` are suitable values. See [Note 3](#).
17. Next initialize the simulation with a spheroid of cells. To do so, we make use of TST’s cell division algorithm. Initialize the simulation with a single, large cell. Set `n_init_cells = 1` and `size_init_cells = 50`. In the TST, cells are initialized

with an Eden growth [41] algorithm that produces disc-shaped growth patterns. `size_init_cells` gives the number of Eden growth cycles. Cleave the cell for a number of times, e.g., by setting `divisions = 7` to create 128 initial cells. Start up the simulation.

18. In the simulation started in Step 17, the initial spheroid quickly expands to satisfy the cells' area constraints. To ensure that this process does not interfere with the observed pattern formation, it is possible to run only the CPM steps for a couple of Monte Carlo Steps (MCS) such that the initial cells 'relax' to their ideal size before starting to chemotact. Set parameter `relaxation = 200`, or the number of MCS that you prefer.

The above simulations use chemotaxis that acts on both *protrusions* and *retractions* [18]. We use these terms to align the chapter with current Cellular Potts terminology, in which a copy attempt that expands a cell into a neighbouring lattice site represents a protrusion, whereas a copy attempt that removes a lattice site from a cell represents a retraction [42]. Earlier editions of this chapter called the same mechanism "extension-retraction" chemotaxis. In the alternative, protrusion-only chemotaxis, only copy attempts that protrude the cell are biased by the chemoattractant gradient [18].

19. In the parameter file prepared in Step 17, set `extension_only = true`. Rerun the simulation. The spheroid will not sprout.
20. Random cell motility can drive sprouting with extension-only chemotaxis due to a curvature effect (see [18] for detail). Increase the cellular temperature to `T = 200`, and rerun the simulation.

An interesting mechanism is that of "passive cell shape changes" [43]. In this mechanism cells elongate as they are 'trapped' by steep chemoattractant gradients they produce themselves. Such steep gradients might, e.g., be due to binding of the secreted morphogens to matrix components [20, 21].

21. Start from a clean copy of `chemotaxis.par`, and release the shape constraint from the endothelial cells, by setting `lambda2 = 0.0`, and run a new simulation. To speed up the simulation reduce the target areas and increase the number of cells. Run the simulation; it should produce rounded clusters.
22. Reduce the width of the chemoattractant gradients, e.g., by reducing the diffusion rate of the chemoattractant to `diff_coeff = 1e-14`. Run the simulation; it should now produced network-like patterns.

3.3 Structure of Tissue Simulation Toolkit programs

So far we have changed only parameter files. To build a new biological model, we now need to see how a model-definition file connects those parameters to the reusable TST framework. We use `src/models/vessel.cpp` as an example; in Section 3.5 we will copy this model and progressively change its biological assumptions.

A basic model, using only the standard model components already implemented in the TST is implemented using a main C++ file defining the simulation. The easiest way to build your own simulation is to edit one of the example models `src/models/sorting.cpp` or `src/models/vessel.cpp`. The main C++ file contains three main code blocks.

```
// Header files included here

INIT {
    // define the initial CPM state
}

TIMESTEP {
    // initialise and advance PDE fields when appropriate
    // perform one CPM Monte Carlo step
    // plot/store output
}

// Model-specific PDE hooks used by ReactionDiffusion()
void PDE::InitialisePDE(CellularPotts *cpm) { ... }
void PDE::InitialiseDiffusionCoefficients(CellularPotts *cpm) { ... }
void PDE::DerivativesPDE(CellularPotts *cpm,
                        PDEFIELD_TYPE *derivs,
                        int x, int y) { ... }

int PDE::MapColour(double val) {
    // optional PDE visualization
}

void Plotter::Plot() {
    // compose PDE and CPM visualization
}

int main(int argc, char *argv[]) {
    // read parameters and start the application
}
```

The INIT code block defines the initial condition of the simulation. The TIMESTEP code calls one Monte Carlo Step, it calls the partial-differential equation solver, and it performs routines that need to be taken care of after each or after a number of Monte Carlo steps, including the visualization. The `main` function initializes the application, and will rarely need to be changed. An additional number of optional functions can be implemented here, including the `PDE::MapColour` functions that is used for visualization of PDEs.

In this subsection we briefly document the main parts of the `src/models/vessel.cpp` model. The file contains additional housekeeping and profiling calls that are not discussed line by line; for example, `PROFILE(...)` marks code regions for optional performance profiling (see [Note 4](#)). For the current class structure and API documentation, refer to the TST 2.0 HTML documentation at <https://www.mathbioleiden.nl/tst-docs/docs/html/sourcecode.html>.

3.3.1 Code block INIT

Here we briefly document the code block INIT of `src/models/vessel.cpp`.

```
INIT {
```

```
    try {
```

The ‘try’ and ‘catch’ statements (below) are an error handling mechanism; it is not necessary to understand it in detail as long as you do not change the structure.

The `CPM->` construction indicates that these are member functions of the class `CellularPotts`; i.e., they read out or change the Cellular Potts plane. To find more information on the use of these functions in the HTML documentation, click “Class Hierarchy”, then “CellularPotts” and look for the name of the function. The `par.` construction indicates that we are taking the value of a parameter defined in the parameter file of the same name.

```
    // Define initial distribution of cells
    /// CPM->GrowInCells(par.n_init_cells,par.size_init_cells,
    ///    par.subfield);
```

`CPM->GrowInCells` sets `n_init_cells` pixels to a spin of increasing value, and expands them by applying an Eden growth algorithm [41] for `size_init_cells` time steps. This will produce approximately disc-shaped initial cells. The cells are distributed over an area of size `size_x/subfield` by `size_y/subfield` centered within the CPM plane.

`GrowInCells` will simply draw cells on the CPM plane. As a next step, we will need to construct an object of class `Cell` for each of the patches. If you want to construct your own initial condition, simply copy `GrowInCells` to your own function, and have it draw cells where you like them. As long as you call `ConstructInitCells` afterwards this will create a viable initial condition.

```
    CPM->ConstructInitCells(*this);
```

Here is a call to the cell division algorithm used in the code introduced in [Step 17](#) to generate an initial spheroid of cells. Section [3.5.5](#) illustrates how to call the `DivideCells` algorithm from within a running simulation to have your cells proliferate. Briefly, a variant of the `DivideCells` function takes a Boolean array `which_cells` that marks the cells for division.

```
    // If we have only one big cell and divide it a few times
```

```

// we start with a nice initial clump of cells.
//
// The behavior can be changed in the parameter file.
for (int i=0;i<par.divisions;i++) {
    CPM->DivideCells();
}

```

At the end of the INIT block we find the closing statement of the try/catch error-handling construct.

```

} catch(const char* error) {
    std::cerr << "Caught exception\n";
    std::cerr << error << "\n";
    std::exit(1);
}

```

3.3.2 Code block TIMESTEP

The code block TIMESTEP implements one time step of the simulation, and is called from the main loop. It typically runs one MCS, runs one or more partial-differential equation steps, and calls the output functions.

```

TIMESTEP {

```

```

    try {

```

‘i’ is a counter that we can increment each time the TIMESTEP is run.

```

        static int i=0;

```

The object ‘dish’ of class ‘Dish’ contains everything we need to run a simulation, i.e., the cellular Potts algorithm, a list of cells, the PDE solver, and so forth.

```

        static Dish *dish=new Dish();

```

The helper class Info collects information from the cells, e.g., by clicking on them interactively.

```

        static Info *info=new Info(*dish, *this);

```

After the relaxation period, TST 2.0 first initializes the PDE state and the spatial diffusion coefficients. On subsequent time steps the model calls `ReactionDiffusion`, which performs one generic diffusion step followed by one model-specific reaction step.

```

    if (i >= par.relaxation) {
        if (i == par.relaxation) {

```

```

    dish->PDEfield->InitialisePDE(dish->CPM);
    dish->PDEfield->InitialiseDiffusionCoefficients(dish->CPM);
} else {
    for (int r = 0; r < par.pde_its; r++) {
        dish->PDEfield->ReactionDiffusion(dish->CPM);
    }
}
}
}

```

The generic `PDE::ReactionDiffusion` implementation in `src/reaction_diffusion/pde.cpp` calls `Diffuse(1)` and then `ForwardEulerStep(1,cpm)`. The latter obtains the reaction terms from the model-specific `PDE::DerivativesPDE` function.

Next we call one MCS of the Cellular Potts model. The PDE field is passed as an argument for the chemotaxis algorithm.

```

dish->CPM->AmoebaeMove(dish->PDEfield);

```

After the CPM step, output is handled by the model's visualization code. Rather than unpacking the older low-level drawing sequence here, we summarize the current TST 2.0 organization in the visualization subsection below. Output is generated every `storage_stride` MCS when either interactive graphics or file storage is enabled.

Increment the time counter.

```

    i++;

} catch(const char* error) {

    cerr << "Caught exception\n";
    std::cerr << error << "\n";
    exit(1);

}

}

```

3.3.3 Model-specific PDE functions and visualization

TST 2.0 separates the generic numerical integration of the PDE from the biological reaction terms. This is an important example of the framework/model distinction introduced above.

Model code. PDE initialization. The supplied `vessel.cpp` initializes the chemical field in `PDE::InitialisePDE`. For the basic vessel model the initial concentration is zero:

```

void PDE::InitialisePDE(CellularPotts *cpm) {

```

```

    for (int x = 0; x < sizex; x++)
        for (int y = 0; y < sizey; y++)
            PDEvars[0][x][y] = 0;
}

```

Model code. Diffusion coefficients. The function `InitialiseDiffusionCoefficients` fills the runtime diffusion-coefficient field. In the basic model it simply copies the parameter value for each layer and lattice site:

```

void PDE::InitialiseDiffusionCoefficients(CellularPotts *cpm) {
    for (int x = 0; x < sizex; x++)
        for (int y = 0; y < sizey; y++)
            for (int l = 0; l < par.n_chem; l++)
                DiffCoeffs[l][x][y] = par.diff_coeff[l];
}

```

Model code. Reaction terms. The biological source and sink terms are implemented by `PDE::DerivativesPDE`. The generic forward-Euler integrator calls this function for every lattice site. In the vessel model, cells secrete the chemoattractant whereas it decays outside the cells:

```

void PDE::DerivativesPDE(CellularPotts *cpm,
                        PDEFIELD_TYPE *derivs,
                        int x, int y) {
    if (cpm->Sigma(x, y)) {
        derivs[0] = par.secr_rate[0];
    } else {
        derivs[0] = -par.decay_rate[0] * PDEvars[0][x][y];
    }
}

```

This implements the non-diffusive terms in the equation introduced in Section 3.2. The generic `ReactionDiffusion` routine combines these model-specific reaction terms with diffusion. We will use the same separation between model code and numerical PDE code in Section 3.5 (see Note 5).

Model code. Visualization. Visualization is composed in `Plotter::Plot`; the model can combine PDE density, CPM cell types, cell boundaries and PDE contour lines without changing the numerical PDE implementation. Function `PDE::MapColour` maps a chemical value to a display colour.

Finally, `main` reads the parameter file, seeds the random-number generator and starts the graphics application; it will normally remain unchanged.

3.4 Implementing the Hamiltonian function: a brief overview over class `CellularPotts`

At this point we cross the boundary from defining a model with existing TST components to extending the framework itself. The current TST 2.0 model API does not yet provide a plug-in interface for arbitrary Hamiltonian terms. Consequently, adding a genuinely new CPM energy contribution such as the haptokinesis term below requires a small, deliberate modification of the `CellularPotts` framework. We mark such steps as **Framework code**; they should remain generic enough to be reusable by other models.

The Cellular Potts algorithm is implemented in class `CellularPotts`; see the Cellular Potts implementation in the TST `src/` tree. In order to add your own components to the Hamiltonian, the two key functions to look for are `CellularPotts::AmoebaeMove`, that implements one Monte Carlo Step, and `CellularPotts::DeltaH` that is responsible for calculating the energy change associated with a potential copy step.

`AmoebaeMove` repeatedly picks a random lattice site with coordinates `x`, `y` and a random adjacent lattice site `xp`, `yp`, and attempts to copy the spin $\sigma(\vec{x})$, `sigma[xp][yp]`, into lattice site `x`, `sigma[x][y]`. Function `DeltaH` calculates the energy change, ΔH , that would result from this potential update. Function `ConvertSpin` actually performs the copy with probability $P(\Delta H) = \{1, \Delta H + H_0 \leq 0; \exp((-\Delta H + H_0)/T), \Delta H + H_0 > 0\}$.

To change the Hamiltonian, change `DeltaH`. The standard implementation includes (in that order) cell adhesion, an area constraint, the chemotaxis algorithm, and the length constraint. These components are switched on and off depending on the parameter values. How to extend the Hamiltonian with additional components, e.g., haptotaxis and haptokinesis, is illustrated in the next Section.

3.5 Implementation of an ECM-guided angiogenesis model using TST

In this section we will build up a model of extracellular matrix (ECM)-guided tumor-induced angiogenesis as described previously [22]. We will add multiple chemical fields and let concentrations in one field influence the secretion or decay of chemicals in another field. The model includes the following rules: 1) Tumors secrete the growth factor VEGF resulting in a VEGF gradient [44]; 2) VEGF stimulates the secretion of diffusive matrix metalloproteinases (MMPs) by endothelial cells (ECs), and 3) MMPs break down ECM components near the cell surface [45, 46]; 4) ECs move chemotactically along VEGF gradients [47, 48], and they 5) migrate toward higher ECM densities (haptotaxis) [49, 50]. 6) Cell speed and spreading are optimal at intermediate ECM densities [51, 52], and 7) cells proliferate if a large part of their surface is in contact with the ECM [53].

We will describe how to implement these steps, specifically how to change the Hamiltonian to model haptokinesis and haptotaxis. We will follow Daub and Merks [22] and add functionality step by step. The aim is not to reproduce a final image in

one jump, but to make the effect of each mechanism visible and testable as the model is assembled.

The model is constructed cumulatively. We first define the vessel geometry and a static VEGF field. We then add the ECM field and haptokinesis, followed by matrix proteolysis, endothelial proliferation, and finally haptotaxis. The six panels of Figure 3 correspond to this sequence. Compile and run the model after each stage before proceeding: the resulting change in collective behaviour provides a direct check on the biological assumption just introduced. It is also instructive to experiment with parameter values after each step.

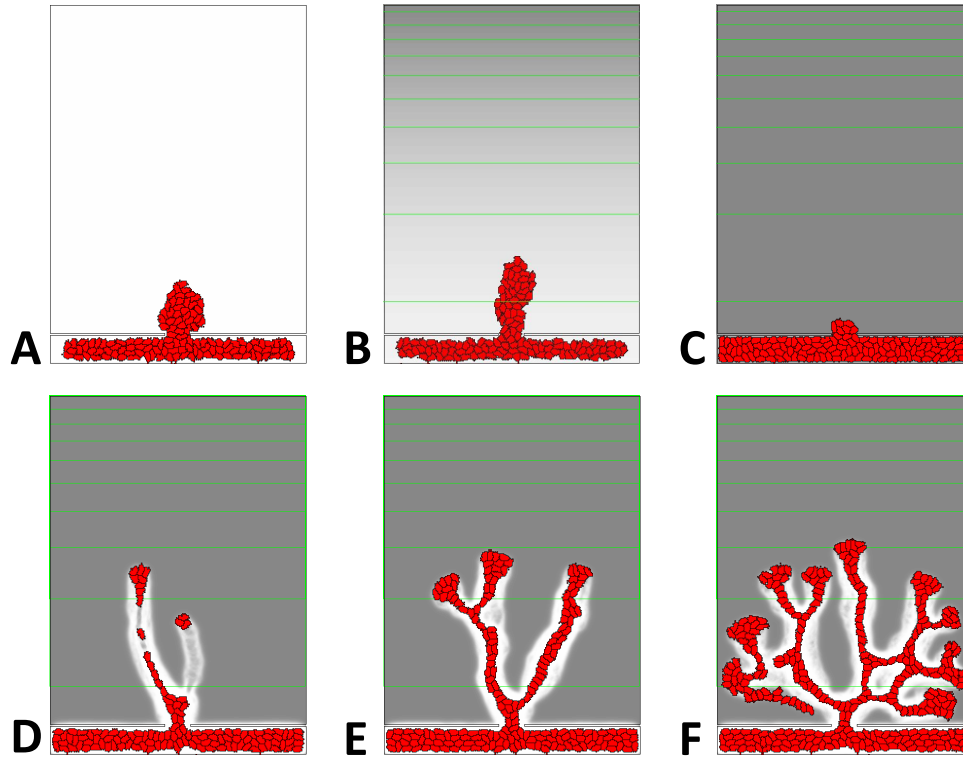


Fig. 3 Typical simulation results after step-by-step implementation of the ECM-guided angiogenesis model (running 35000 MCS), starting with only adhesion energy and area and length constraints (A), adding chemotaxis (B), haptokinesis (C), proteolysis (D), proliferation (E) and haptotaxis (F). Background coloring reflects VEGF concentration (B) or ECM density (C–F); the green contour lines represent the VEGF gradient.

23. Because this exercise will eventually add a small number of reusable framework extensions as well as a new model, create a dedicated Git branch before starting:

```
git checkout -b ecm-model
```

There is no need to copy the complete TST source tree; Git will keep the model and framework changes isolated and make them easy to inspect or undo (see [Note 6](#)).

24. We will use `src/models/vessel.cpp` and `data/chemotaxis.par` as templates. From the top-level TST directory, make the two copies explicitly:

```
cp src/models/vessel.cpp src/models/ecm.cpp
cp data/chemotaxis.par data/ecm.par
```

From this point onward, edit the new `ecm` files rather than the original `vessel` model.

25. TST 2.0 has a separate `qmake` project file for each C++ model. Copy the project file for `vessel` to a new project file for `ecm`:

```
cp src/vessel.pro src/ecm.pro
```

Open `src/ecm.pro` in a text editor. Replace `vessel` by `ecm` in both model-specific entries: the target name and the source-file name. Leave the shared TST settings unchanged.

26. Register the new model in the top-level `Makefile`. Find the list of standard model targets containing `bin/vessel` and add `bin/ecm` to this list.
27. Build the new model from the top-level TST directory with:

```
make bin/ecm
```

The root `Makefile` will run `qmake` and make for this model and places the executable in `bin/ecm`. Test the model from the `bin/` directory with the new parameter file, e.g.:

```
./ecm ../data/ecm.par
```

This command can be run from the `bin/` directory as shown. In the TST 2.0 version used for this chapter, input resources such as `default.ctb` and the J-table are resolved relative to the parameter file, so their location does not depend on the directory from which the executable is started.

3.5.1 Dish setup: cells behind a vessel wall

We will first set up the initial situation: ECs are placed in a parent vessel at the bottom of a rectangular dish. A two-pixel-thick vessel wall separates the vessel from the tissue, except for a small central gap through which the cells can migrate (Figure 4A). Fixed walls are a generic feature of TST: the model specifies *where* the wall is, while the Cellular Potts framework ensures that fixed wall sites do not participate in copy attempts.

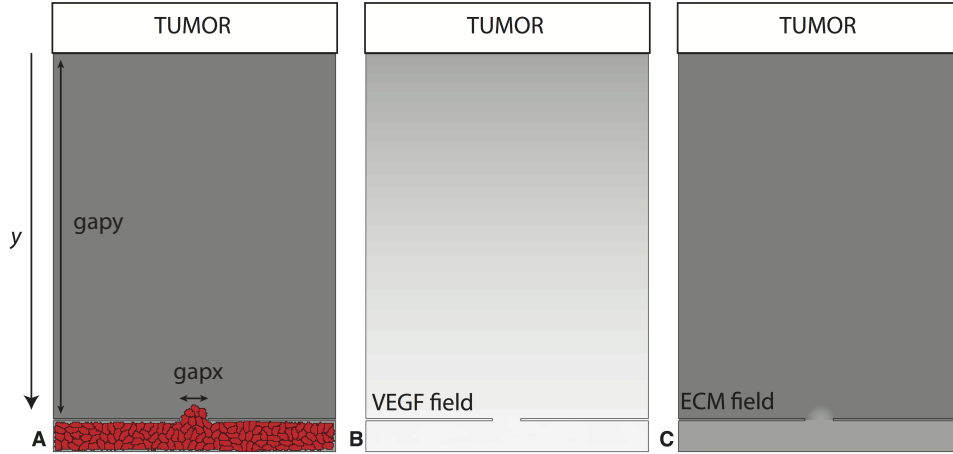


Fig. 4 Setup of model domain. A Initial ECM field (grey) and model configuration: ECs are placed behind a vessel wall and can migrate through a gap in the wall. B Steady state VEGF field; C Initial ECM field. Adapted from Figure 1 in [22].

28. **Parameter definition.** Introduce two new parameters, `gapx` and `gapy`. The parameter `gapx` defines the width of the opening and `gapy` the vertical position of the lower row of the vessel wall. Add the following lines to `src/parameters/parameters.hpp` in a dedicated `SECTION("Daub and Merks model")` block (see [Note 7](#)):

```
SECTION("Daub and Merks model")
PARAMETER(int, gapx, 0, "Width of the gap in the vessel wall")
PARAMETER(int, gapy, 0, "Vertical position of the vessel wall")
```

Here, `PARAMETER(type, name, default, description)` declares a setting that becomes a member of the global `par` parameter object and can be read from the parameter file. Unlike older versions of TST, no corresponding declaration needs to be added to a separate parameter header: the `PARAMETER` macros generate the members of the global `Parameter` object and take care of reading and writing them.

29. **Model code.** Construct the vessel wall in the `INIT` block of `src/models/ecm.cpp`, before growing the endothelial cells. TST represents a fixed wall site by the special state $\sigma = -1$; `CellularPotts::SetWall` provides a model-level interface for creating such sites. Next, use the general six-argument version of `GrowInCells` to place the initial cells in the vessel compartment below the wall. Note that the wall occupies rows `gapy` and `gapy+1`, so the cell compartment starts at `gapy+2`.

```
INIT {
  try {
```

```

// Construct a two-pixel-thick vessel wall,
// leaving a central gap.
if (par.gapy) {
    for (int x=1; x<(par.size_x-par.gap_x)/2; x++) {
        CPM->SetWall(x, par.gapy);
        CPM->SetWall(x, par.gapy+1);
    }
    for (int x=(par.size_x+par.gap_x)/2;
         x<par.size_x-1; x++) {
        CPM->SetWall(x, par.gapy);
        CPM->SetWall(x, par.gapy+1);
    }
}

// Grow endothelial cells in the vessel compartment.
int sx = static_cast<int>((par.size_x-2)/par.subfield);
int sy = par.size_y-3-par.gapy;
int offset_x = (par.size_x-2-sx)/2;
int offset_y = par.gapy+2;

CPM->GrowInCells(par.n_init_cells, par.size_init_cells,
                 sx, sy, offset_x, offset_y);
CPM->ConstructInitCells(*this);

// If requested, divide the initial cells.
for (int i=0; i<par.divisions; i++)
    CPM->DivideCells();

// Build the edge list only after the complete
// initial geometry has been constructed.
CPM->InitialiseEdgeList();

} catch (const char *error) {
    cerr << "Caught exception\n";
    std::cerr << error << "\n";
    exit(1);
}
}

```

The existing Eden-growth algorithm preserves fixed wall sites, so no ECM-specific changes to `GrowInCells` are required.

30. Set the parameters in `data/ecm.par`. At this stage no dynamic chemical field is required. Set `pde_its=0` so that the PDE solver is skipped while we test only the Cellular Potts geometry and migration.

```

# Cellular Potts parameters
T = 100
target_area = 50
target_length = 15
lambda = 25
lambda2 = 25
Jtable = Jecm.dat
conn_diss = 2000
vecadherinknockout = false
chemotaxis = 0
border_energy = 100
extensiononly = true

neighbours = 3
periodic_boundaries = false

# PDE parameters
n_chem = 1
diff_coeff = 1e-13
decay_rate = 1.8e-4
secre_rate = 1.8e-4
saturation = 0
dt = 2.0
dx = 2.0e-6
pde_its = 0

# initial conditions
n_init_cells = 125
size_init_cells = 13
sizex = 250
sizey = 350
gapx = 25
gapy = 320

divisions = 0
mcs = 40000
rseed = -1
subfield = 1.05
relaxation = 0

# output
storage_stride = 10
graphics = true
store = false
datadir = ../data_film

```

31. Rather than changing the standard interaction table, make a model-specific copy for the ECM tutorial:

```
cp ../data/J.dat ../data/Jecm.dat
```

Change the energy settings in `Jecm.dat` to:

```
2
0
25 40
```

The parameter file above therefore refers to `Jtable = Jecm.dat`.

32. Compile and run the application, making sure that you save all files changed in this section. You should now see the ECs in the vessel compartment migrating through the central opening. Change `gapx` to test the effect of the gap size. Because `pde_its=0`, this first simulation tests only the CPM part of the model and should run substantially faster than later stages that contain a dynamic PDE. Compare this initial stage with Figure 3A.

3.5.2 Dish setup: Static VEGF gradient

At the top of the dish a tumor secretes VEGF. The ECs can migrate through the gap in the parent vessel wall and move chemotactically up the VEGF gradient toward the tumor. In our model we assume a planar steady-state VEGF gradient (Figure 4B) given by:

$$c_v(x_2) = c_v(0)e^{-x_2/\lambda} \text{ with } \lambda = \sqrt{D/\epsilon} \quad (5)$$

where $c_v(x_2)$ is the VEGF concentration at distance y from the tumor, and D and ϵ are the diffusion coefficient and decay rate of VEGF. We will initialize the gradient once and it will not further change during the simulation.

33. **Parameter definition.** Set up the VEGF gradient using the diffusion and degradation parameters (D , `diff_coeff` and ϵ , `decay_rate`) as well as the VEGF concentration at the tumor, $c_v(0)$. We create a new parameter `init_conc`, to set this VEGF concentration in the parameter file. As this parameter will also be used to initialize other PDE layers, it is a vector. We also create a vector parameter `chem_type`, which defines the chemical type for each layer (VEGF, ECM, MMP, or others). Add the following lines to `src/parameters/parameters.hpp`, after the definition of `n_chem`:

```
PARAMETER(std::vector<double>, chem_type, {0.0},
           "Chemical type represented by each PDE layer")
CONSTRAINT(chem_type.size() == n_chem,
           "Number of chem_type values does not match n_chem")

PARAMETER(std::vector<double>, init_conc, {0.0},
```

```

        "Initial concentration for each PDE layer")
CONSTRAINT(init_conc.size() == n_chem,
           "Number of init_conc values does not match n_chem")

```

CONSTRAINT(expression, message) adds a consistency check that is evaluated when the parameters are validated. If the expression is false, TST exits with the specified error message. Thus the constraints above ensure that the vectors contain one value for every PDE layer.

34. Define the first PDE layer for the VEGF concentrations in the dish. As our model will have different PDE layers, each with a different chemical, we will annotate each layer which makes it easier to read and to change the code. In `src/models/ecm.cpp` add the VEGF chemical and layer near the beginning of the file.

```

// Chemicals in ecm application
const int VEGF = 0;

// The layers in ecm application
int L_VEGF = -1;

```

```
using namespace std;
```

35. Add the following code to the INIT block in `src/models/ecm.cpp`, to give each layer the proper name. In the parameter file we will define the chemical type for each layer. In this way you do not necessarily need to define all 3 layers and you are free to change the order.

```

INIT {
    (...)

    // If requested, divide the initial cells.
    for (int i=0;i<par.divisions;i++) {
        CPM->DivideCells();
    }

    // Assign layer indices to specific layer variables
    for (int l=0;l<PDEfield->Layers();l++){
        switch ((int) par.chem_type[l]) {
            case VEGF: L_VEGF=l; break;
        }
    }

    (...)
}

```

36. Add code in `src/models/ecm.cpp` to initialize the VEGF PDE field. By default, other layers will be initialized with a uniform chemical density (`init_conc`). Later on we will add code to set up the ECM field

```
void PDE::InitialisePDE(CellularPotts *cpm) {
    for (int x=0;x<par.size_x;x++)
        for (int y=0;y<par.size_y;y++)
            for (int l=0;l<Layers();l++){

                if (l==L_VEGF) {

                    double D = par.diff_coeff[l];
                    double d = par.decay_rate[l];
                    double b, conc;

                    b = -1.0/sqrt(D/d);
                    // Calculate gradient, when concentration at 'top'
                    // of dish is given
                    conc = par.init_conc[l]*exp(b*y*par.dx);
                    setValue(l,x,y,conc);

                } else {
                    /* =====
                     /  OTHER LAYERS: UNIFORM DENSITY
                     =====*/
                    setValue(l,x,y,par.init_conc[l]);
                }
            }
}
```

37. **Model code.** The VEGF profile is an analytical, static field. We nevertheless keep the physical VEGF diffusion coefficient in `par.diff_coeff`, because it determines the length scale $\lambda = \sqrt{D/\epsilon}$ used to initialize the profile. To prevent the generic PDE solver from subsequently diffusing this layer, set its *runtime* diffusion coefficient to zero in the model-specific `PDE::InitialiseDiffusionCoefficients` function in `src/models/ecm.cpp`:

```
void PDE::InitialiseDiffusionCoefficients(CellularPotts *cpm) {
    for (int x = 0; x < size_x; x++)
        for (int y = 0; y < size_y; y++)
            for (int l = 0; l < par.n_chem; l++)
                DiffCoeffs[l][x][y] = (l == L_VEGF) ? 0.0
                                          : par.diff_coeff[l];
}
```

No change to the generic `PDE::Diffuse` function is required. This is precisely the distinction between a model choice (this particular VEGF field is static) and generic numerical machinery (how a diffusing field is integrated).

38. **Model code.** During this stage the model has no dynamic reaction terms, so `DerivativesPDE` sets all derivatives to zero:

```
void PDE::DerivativesPDE(CellularPotts *cpm,
                        PDEFIELD_TYPE *derivs,
                        int x, int y) {
    for (int l = 0; l < layers; l++)
        derivs[l] = 0.0;
}
```

Because the VEGF profile is analytical and static, keep `pde_its=0` in this section. There is no reason to traverse the complete PDE lattice at every MCS when neither diffusion nor reaction is being integrated. We will switch PDE stepping on when the dynamic MMP field is introduced in Section 3.5.4.

39. Set the VEGF parameters in the parameter file `ecm.par` in the `data` folder.

```
# Cellular Potts parameters
chemotaxis = 6000

# PDE parameters
chem_type = 0
init_conc = 0.87
diff_coeff = 6e-11
decay_rate = 1e-3
secre_rate = 0
```

40. Compile and run the application. You will now see the vessel growing slowly towards the tumor at the top of the dish. You can see the effect of the `chemotaxis` parameter by setting it to different values. Setting it to zero provides the no-chemotaxis control; compare the resulting behaviour with Figure 3B. Increasing the chemotaxis coefficient will generally increase the speed with which the vessel grows towards the tumor. As the simulation is rather time consuming, you could change the output parameters to save the graphic to a file every 1000 MCS. Make sure you have created a folder named `data_film` in the top-level TST directory. See Note 8. For running TST without an interactive display, see Note 9.

```
# output
storage_stride = 1000
graphics = true
store = true
datadir = ../data_film
```

3.5.3 Adding interaction with the extracellular matrix: haptokinesis

In the coming sections we will add the effect of interactions between ECs and the ECM to the model. We will start with including **haptokinesis** in our simulation program, the phenomenon that cells have optimal motility at intermediate ECM densities.

41. **Parameter definition. Framework code.** Add a vector parameter **haptokinesis** to define the haptokinetic strength for each PDE layer. Keep the existing scalar **chemotaxis** parameter unchanged; at this stage chemotaxis still acts only on the VEGF layer. Also introduce **min_conc** and **max_conc** for the concentration range of each PDE layer, and **cont_color** and **grad_color** for visualization. In `src/parameters/parameters.hpp`, add the following definitions in the `SECTION("Daub and Merks model")` block (see [Note 10](#) and [Note 11](#)):

```
PARAMETER(std::vector<double>, haptokinesis, {0.0},
           "Haptokinesis strength for each PDE layer")
CONSTRAINT(haptokinesis.size() == n_chem,
            "Number of haptokinesis values does not match n_chem")

PARAMETER(std::vector<double>, min_conc, {0.0},
           "Minimum concentration for each PDE layer")
PARAMETER(std::vector<double>, max_conc, {99.0},
           "Maximum concentration for each PDE layer")
PARAMETER(std::vector<double>, cont_color, {7.0},
           "Contour colour for each PDE layer; 0 disables contours")
PARAMETER(std::vector<double>, grad_color, {1.0},
           "Nonzero value selects a PDE layer for gradient display")

CONSTRAINT(min_conc.size() == n_chem,
            "Number of min_conc values does not match n_chem")
CONSTRAINT(max_conc.size() == n_chem,
            "Number of max_conc values does not match n_chem")
CONSTRAINT(cont_color.size() == n_chem,
            "Number of cont_color values does not match n_chem")
CONSTRAINT(grad_color.size() == n_chem,
            "Number of grad_color values does not match n_chem")
```

TST 2.0 already uses `std::vector<double>` for its vector-valued PDE parameters. We use the same type here for **chem_type**, **cont_color**, and **grad_color**; values that act as integer identifiers are cast to `int` when used.

42. Add a new pde layer containing the ECM. In `src/models/ecm.cpp` add the following:

```
// Chemicals in ecm application
```

```
const int VEGF = 0;
const int ECM = 1;
```

```
// The layers in ecm application
int L_VEGF = -1;
int L_ECM = -1;
```

The L... variables store the PDE-layer number associated with each chemical type; -1 means that the corresponding layer has not yet been assigned.

43. **Model code.** In the INIT block in `src/models/ecm.cpp`, add:

```
INIT {
    (...)

    // Assign layer indices to specific layer variables
    for (int l=0;l<PDEfield->Layers();l++){
        switch ((int) par.chem_type[l]) {
            case VEGF: L_VEGF=l;break;
            case ECM: L_ECM=l;break;
        }
    }
    (...)
}
```

44. Define the initial concentrations of the ECM field in the dish. There will be a uniform concentration of `init_conc` outside the vessel, an intermediate concentration inside the vessel, and a half-circular gradient around the gap outside the vessel as transition between the concentrations inside and outside the vessel (Figure 4C).

```
INIT {

    try {

        (...)

        // Initialisation of PDE fields
        for (int x=0;x<par.size_x;x++)
        for (int y=0;y<par.size_y;y++)
            for (int l=0;l<PDEfield->Layers();l++){
                if (l==L_VEGF) {
                    (...)

                    /* =====
```

```

/ UNIFORM ECM DENSITY
/ Uniform ECM density outside vessel
/ Intermediate density inside vessel
/ Behind gap circular gradient as transition to outside
=====*/
} else if (l==L_ECM) {
    // If there is no wall uniform density
    if (!par.gapy)
        PDEfield->setValue(l,x,y,par.init_conc[l]);

    // If there is a wall and (x,y) lies behind it,
    // set to intermediate concentration
    else {
        double con = (par.max_conc[l]-par.min_conc[l])/2.;
        if (y > par.gapy)
            PDEfield->setValue(l,x,y,con);
        // If there is a wall and (x,y) lies outside,
        // set to outside density or (when near gap)
        // 'in between' density
        else {
            double dist2 = (x-par.size_x/2)*(x-par.size_x/2) +
                (y-par.gapy)*(y-par.gapy);
            double dist2max = (par.gap_x)*(par.gap_x)/4.;
            // If (x,y) is outside half circle around gap
            // set to uniform outside density
            if (dist2 > dist2max)
                PDEfield->setValue(l,x,y,par.init_conc[l]);
            else
                PDEfield->setValue(l,x,y,con +
                    (par.init_conc[l]-con)*dist2/dist2max);
        }
    }
} else {
    (...)
}
(...)
}

```

45. **Model code.** The ECM field, like the VEGF field, is non-diffusive. Extend `InitialiseDiffusionCoefficients` so that both static layers have zero runtime diffusion coefficient:

```

void PDE::InitialiseDiffusionCoefficients(CellularPotts *cpm) {

```

```

    for (int x = 0; x < sizex; x++)
        for (int y = 0; y < sizey; y++)
            for (int l = 0; l < par.n_chem; l++)
                DiffCoeffs[l][x][y] =
                    (l == L_VEGF || l == L_ECM) ? 0.0 : par.diff_coeff[l];
}

```

Again, the generic `Diffuse` implementation remains untouched.

We will next add haptokinesis to the model, the phenomenon that cell motility depends on the concentration of ECM, with highest motility at intermediate ECM densities. We define the haptokinesis energy term as:

$$\Delta H_{\text{haptokinesis}} = \eta \left(-1 + \frac{1}{\rho\sqrt{2\pi}} \exp \left(\frac{-(\delta(\sigma(\vec{x}), 0)c_E(\vec{x}') - \mu)^2}{2\rho^2} \right) \right) \quad (6)$$

where η is the haptokinesis strength and $\mu = \frac{1}{2}$ is the intermediate ECM density; ECM densities in the model are in the range $c_E \in [0, 1]$. The standard deviation ρ is set to a value $\rho = 0.2$ to ensure diversity of cell motility over the range of available ECM values. In the implementation used for the simulations in Daub and Merks [22], copy attempts that are not extensions into the ECM receive the maximal haptokinetic penalty; we retain that implementation here so that the chapter follows the published model (see [Note 12](#)).

46. **Framework code.** In `src/cellular_potts/ca.cpp` add the new Haptokinesis function:

```

int CellularPotts::DeltaH(int x, int y, int xp, int yp, PDE *PDEfield,
                          AdhesionDisplacements *adh_disp)
{
    (...)
}

// Haptokinesis
// Cells have optimal motility at intermediate ECM densities
int CellularPotts::Haptokinesis(int x, int y, int xp, int yp,
                                PDE *PDEfield)
{
    double s = 0.2;
    double m = 0.5;
    double c = 0.0;
    int tmpDH = 0;

    for (int l=0; l<PDEfield->Layers(); l++){
        if (par.haptokinesis[l]) {

```

```

    //Only during extension, look at concentration at cell site
    if (sigma[x][y])
        //if no extension, set c to min_conc to have max penalty
        c = par.min_conc[1];
    else
        c = PDEfield->get_PDEvars(1,xp,yp);

    double c_scaled = (c-par.min_conc[1])/(par.max_conc[1] -
        par.min_conc[1]);
    tmpDH += par.haptokinesis[1] *
        (-1+exp(-1*(c_scaled-m)*(c_scaled-m)/(2*s*s))/
        sqrt(2*M_PI*s*s));
}
}
return tmpDH;
}

```

47. **Framework code.** In `src/cellular_potts/ca.hpp` add the (private) Haptokinesis function:

```

class CellularPotts {
    (...)
private:
    (...)
    int DeltaH(int x, int y, int xp, int yp, PDE *PDEfield,
        AdhesionDisplacements *adh_disp);
    int Haptokinesis(int x, int y, int xp, int yp, PDE * PDEfield);
    (...)
};

```

48. In `CellularPotts::DeltaH` in `src/cellular_potts/ca.cpp`, leave the existing chemotaxis code unchanged and add the call to Haptokinesis immediately after it:

```

int CellularPotts::DeltaH(int x, int y, int xp, int yp, PDE *PDEfield,
    AdhesionDisplacements *adh_disp)
{
    (...)
    /* existing scalar chemotaxis code remains here */
    (...)

    // Haptokinesis
    // Cells have optimal motility at intermediate ECM densities
    if (PDEfield)
        DH -= Haptokinesis(x,y,xp,yp,PDEfield);
}

```

```
(...)  
}
```

Set the ECM layer to be displayed after it has been introduced. With the layer ordering VEGF, ECM, this is layer 1; in the complete three-layer model the same ECM layer remains layer 1. The parameters used below therefore select ECM as the density field and enable its contour display where desired. We will use the `Plotter` layer-selection interface to select the PDE layer indicated by `grad_color` for density display and the layer indicated by `cont_color` for contour lines. The TST 2.0 `Plotter` methods `SetPDEDensityLayer` and `SetPDEContourLayer` provide these selections; the common plotting path is then used for both on-screen and stored output.

First create two new variables `L_plotgrad` and `L_plotcont` and assign them the proper layer index in the INIT block:

```
// Which layers to plot?  
int L_plotgrad = -1;  
int L_plotcont = -1;  
  
using namespace std;  
  
INIT {  
    try {  
        (...)  
  
        // Assign layer indices to specific layer variables  
        for (int l=0;l<PDEfield->Layers();l++){  
            (...)  
        }  
  
        // Assign layer indices to plot variables  
        for (int l=0;l<PDEfield->Layers();l++){  
            if (par.cont_color[l]) L_plotcont = l;  
            if (par.grad_color[l]) L_plotgrad = l;  
        }  
        (...)  
    }  
    (...)  
}
```

Next, change the `Plotter::Plot` to plot the specified a gradient and contour lines in the specified layers:

```
void Plotter::Plot() {  
    graphics->BeginScene();  
    graphics->ClearImage();
```

```

    if (L_plotgrad > -1) {
        SetPDEDensityLayer(L_plotgrad);
        plotPDEDensity();
    }
    if (L_plotcont > -1) {
        SetPDEContourLayer(L_plotcont);
        plotPDEContourLines();
    }

    plotCPMCellTypes();
    plotCPMLines();

    graphics->EndScene();
}

```

49. In the parameter file `ecm.par` we add and change the following parameters:

```

# PDE parameters
n_chem = 2
chem_type = 0,1
init_conc = 0.87,0.9
diff_coeff = 6e-11,0
decay_rate = 1e-3,0
secre_rate = 0,0
chemotaxis = 6000
haptokinesis = 0,200
min_conc = 0,0
max_conc = 1,1
cont_color = 1,0
grad_color = 0,1

```

Setting `cont_color = 1,0` will show contour lines for the first layer, the VEGF field; `grad_color = 0,1` will plot the density of the second layer, the ECM field. To switch off plotting of contour or density plots, set these vectors to all zeros.

50. Compile and run the code. You will now see that the cells will not migrate far out of the vessel, because the ECM concentrations are too high. To isolate the effect of haptokinesis, repeat the simulation with `haptokinesis = 0,0` and compare the result with the standard value `haptokinesis = 0,200`; compare the resulting behaviour with Figure 3C.

3.5.4 Introducing interaction between layers: adding proteolysis to the model

In this section we add proteolysis: degradation of ECM by matrix metalloproteinases (MMPs). ECs secrete MMPs at a rate that depends on the local VEGF concentration,

and MMPs in turn degrade the ECM. These are biological assumptions of the Daub and Merks model, so we implement the reaction terms directly in the model-specific code rather than introducing generic secretion and degradation tables into the TST framework.

51. **Parameter definition.** In the `SECTION("Daub and Merks model")` block of `src/parameters/parameters.hpp`, add a parameter for MMP-dependent ECM degradation. MMP secretion and decay use the existing vector-valued `secre_rate` and `decay_rate` parameters, respectively:

```
PARAMETER(double, ECMdegradation, 2e-3,
           "MMP-dependent ECM degradation rate")
```

52. Add the MMP chemical and its layer. In `src/models/ecm.cpp`, extend the chemical identifiers and layer variables to:

```
// Chemicals in ecm application
const int VEGF = 0;
const int ECM = 1;
const int MMP = 2;

// The layers in ecm application
int L_VEGF = -1;
int L_ECM = -1;
int L_MMP = -1;
```

53. In the `INIT` block in `src/models/ecm.cpp`, extend the layer-assignment code so that it also recognizes MMP:

```
// Assign layer indices to specific layer variables
for (int l=0; l<PDEfield->Layers(); l++) {
    switch ((int) par.chem_type[l]) {
        case VEGF: L_VEGF=l; break;
        case ECM:  L_ECM=l;  break;
        case MMP:  L_MMP=l;  break;
    }
}
```

MMPs degrade the ECM at a rate proportional to the local MMP and ECM concentrations:

$$\frac{\partial c_E(\vec{x}, t)}{\partial t} = -\delta(\sigma(\vec{x}), 0) \epsilon_{EM} c_M(\vec{x}, t) c_E(\vec{x}, t), \quad (7)$$

where c_E and c_M denote ECM and MMP concentrations and ϵ_{EM} is the ECM degradation constant.

ECs secrete MMPs at a rate proportional to local VEGF:

$$\frac{\partial c_M(\vec{x}, t)}{\partial t} = \alpha_{MV} c_V(\vec{x}, t) (1 - \delta(\sigma(\vec{x}), 0)) H(c_{M, \max} - c_M(\vec{x}, t)) - \delta(\sigma(\vec{x}), 0) \epsilon_M c_M(\vec{x}, t) + D_M \nabla^2 c_M(\vec{x}, t), \quad (8)$$

where c_V is the local VEGF concentration, α_{MV} is the MMP secretion rate, ϵ_M is the MMP decay rate and D_M is the MMP diffusion coefficient. Cells secrete MMP only at lattice sites they cover, whereas ECM degradation and MMP decay occur in the medium. The maximum MMP concentration is set by `max_conc`.

54. **Model code.** Implement these VEGF–MMP–ECM reaction terms in the model-specific PDE::DerivativesPDE function in `src/models/ecm.cpp`:

```
void PDE::DerivativesPDE(CellularPotts *cpm,
                        PDEFIELD_TYPE *derivs,
                        int x, int y) {
    for (int l=0; l<layers; l++)
        derivs[l] = 0.0;

    if (cpm->Sigma(x,y) > 0) {
        if (PDEvars[L_MMP][x][y] < par.max_conc[L_MMP])
            derivs[L_MMP] =
                par.secr_rate[L_MMP] * PDEvars[L_VEGF][x][y];
    } else {
        derivs[L_ECM] =
            -par.ECMdegradation * PDEvars[L_MMP][x][y]
            * PDEvars[L_ECM][x][y];
        derivs[L_MMP] =
            -par.decay_rate[L_MMP] * PDEvars[L_MMP][x][y];
    }
}
```

The standard `ReactionDiffusion` call in the `TIMESTEP` block invokes this function, so no separate secretion call is needed. If your copied `vessel.cpp` template still contains a model-specific `PDE::Secrete` implementation, leave its body empty for the `ecm` model so that the reaction terms are not applied twice.

55. Extend `ecm.par` to three PDE layers. Keep the VEGF concentration, chemotaxis and haptokinesis values used in the preceding steps:

```
# PDE parameters
n_chem = 3
chem_type = 0,1,2
init_conc = 0.87,0.9,0
diff_coeff = 6e-11,0,1e-14
decay_rate = 0,0,1e-3
secr_rate = 0,0,8e-5
```

```

pde_its = 15
chemotaxis = 6000
haptokinesis = 0,200,0
min_conc = 0,0,0
max_conc = 1,1,1
cont_color = 0,0,0
grad_color = 0,1,0

```

```
ECMdegradation = 2e-3
```

This is the first stage in which a PDE layer evolves dynamically, so `pde_its` is getting a positive value (15) now. The additional reaction–diffusion calculations make this stage noticeably slower than the preceding static-field simulations.

56. Compile and run the code. You should now see the ECM layer being degraded by MMP secreted by the cells. Some cells may split off and migrate toward the tumor, while the sprout itself still has limited growth. Setting `ECMdegradation = 0` provides a simple control for the proteolysis mechanism. Compare this stage with Figure 3D.

3.5.5 Adding proliferation to create a growing sprout

By adding proteolysis we were able to create a growing and branching sprout, but it was unable to reach the tumor. For this we add proliferation to the model. We let cells divide, when their proportion of cell surface NOT touching other cells is larger than a certain threshold.

57. First add three new parameters: `allowdiv`, which allows proliferation if set to true, `growthrate`, which determines the rate with which the cells grow, and `ECMsurfratio`, which is the minimum proportion of the cell surface connecting with the ECM required to allow division. Add the following lines to `src/parameters/parameters.hpp`. Only the three model-specific parameters below need to be added; `border_energy` is already a standard TST parameter.

```

PARAMETER(bool, allowdiv, false,
           "Whether to allow ECM-dependent endothelial cell division")
PARAMETER(int, growthrate, 2,
           "Increment of target cell size during growth")
PARAMETER(double, ECMsurfratio, 0.6,
           "Minimum cell-ECM surface ratio required for division")

```

58. **Framework code.** Create the new functions `GrowAndDivideECs` and `MeasureCellSurfaces` in `src/cellular_potts/ca.cpp`. The first is an overload of the existing growth-and-division routine: C++ allows functions with the same

name when their argument lists differ. This version makes proliferation dependent on the amount of cell-ECM contact. `MeasureCellSurfaces` measures, for each cell, its total boundary and the part contacting ECM:

```
void CellularPotts::GrowAndDivideCells(int growth_rate) {...}

// Grow cells and divide those cells (with certain probability)
// when ratio (cell-ECM surface)/(total surface) is above threshold
void CellularPotts::GrowAndDivideECs(int growth_rate,
                                     double ECMSurfratio) {

    //calculate length growth rate
    double growth_rate_length;
    growth_rate_length = ((double)par.target_length/2)/
        (((double) par.target_area/2)/growth_rate);
    MeasureCellSurfaces();

    vector<Cell>::iterator c=cell->begin(); ++c;
    vector<bool> which_cells(cell->size());

    for (;c!=cell->end(); c++) {
        c->SetTargetArea(min(par.target_area,c->
            TargetArea()+growth_rate));
        c->SetTargetLength(min((double)par.target_length,
            c->TargetLength()+growth_rate_length));
        double r = (double)c->ECMSurface()/c->TotSurface();
        if ((c->Area())>=par.target_area) && r > ECMSurfratio) {
            if (RANDOM()< r-ECMSurfratio)
                which_cells[c->Sigma()]=true;
            else
                which_cells[c->Sigma()]=false;
        } else {
            which_cells[c->Sigma()]=false;
        }
    }
    DivideCells(which_cells);
}

// Measure per cell the ratio (cell-ECM surface)/total surface
// When this value is above par.ECMSurfratio, the cell can divide
void CellularPotts::MeasureCellSurfaces(void){
    // Clean ECMSurfaces of all cells, including medium
    for (vector<Cell>::iterator c=cell->begin();c!=cell->end();c++) {
        c->SetECMSurface(0);
    }
}
```

```

        c->SetTotSurface(0);
    }

    for (int x=1;x<sizeX-1;x++)
    for (int y=1;y<sizeY-1;y++) {
        if (sigma[x][y]>0) {
            for (int i=1;i<=4;i++) {
                int x2,y2;
                x2=x+nx[i]; y2=y+ny[i];
                if ((!sigma[x2][y2]) &&
                    (par.gapy==0 || y2 < par.gapy-par.target_length)){
                    (*cell)[sigma[x][y]].IncrementTotSurface();
                    (*cell)[sigma[x][y]].IncrementECMSurface();
                }
                else if ((sigma[x2][y2]!=sigma[x][y]))
                    (*cell)[sigma[x][y]].IncrementTotSurface();
            }
        }
    }
}

```

Add the functions to `src/cellular_potts/ca.hpp`:

```

class CellularPotts {
    (...)
public:
    (...)
    void GrowAndDivideCells(int growth_rate);
    void GrowAndDivideECs(int growth_rate, double ECMSurfratio);
    void MeasureCellSurfaces(void);
    (...)
};

```

59. In `src/cellular_potts/cell.hpp` add new functions and variables for calculating the total cell surface and the cell surface touching the ECM:

```

class Cell
{
    (...)
public:
    (...)
    inline void SetECMSurface(int s){ecm_surf = s;}
    inline int ECMSurface() {return ecm_surf;}
    inline void IncrementECMSurface(){ecm_surf++;}

```

```

inline void SetTotSurface(int s){tot_surf = s;}
inline int TotSurface() {return tot_surf;}
inline void IncrementTotSurface(){tot_surf++;};
(...)

```

protected:

```

int ecm_surf; //cell surface connected to ECM
int tot_surf; //total cell surface

```

60. Add a call to the `GrowAndDivideECs` function to `src/models/ecm.cpp` in the `TIMESTEP` block:

```

TIMESTEP {
  (...)

  dish->CPM->AmoebaeMove(dish->PDEfield);

  // Grow cells and divide those with a ratio
  // (cell-ECM surface)/(total surface) above threshold
  // ECMSurfratio
  if (par.allowdiv && i>=par.relaxation) {
    if (!(i%5)) {
      dish->CPM->GrowAndDivideECs(par.growthrate,
                                par.ECMSurfratio);
    }
  }
  (...)
}

```

61. In the parameter file, set the following values:

```

extensiononly = true

# cell growth and division
allowdiv = true
growthrate = 2
ECMSurfratio = 0.73

```

Compile and run the model at this point. Compared with the proteolysis-only stage, the sprout should now be able to grow further because cells with sufficient cell-ECM contact can divide (Figure 3E).

3.5.6 One more ECM interaction: adding haptotaxis to the model

The last ingredient we add is *haptotaxis*, directed movement of cells up gradients of ECM density. The haptotaxis energy term is:

$$\Delta H_{\text{haptotaxis}} = -\Gamma \delta(\sigma(\vec{x}), 0) \left(\frac{c_E(\vec{x})}{1 + s c_E(\vec{x})} - \frac{c_E(\vec{x}')}{1 + s c_E(\vec{x}')} \right), \quad (9)$$

where Γ is the haptotactic strength and s is a saturation parameter. The saturation term reduces the response at high ECM concentrations.

Haptotaxis has the same mathematical form as chemotaxis, but the completed model needs a separate taxis strength and saturation value for each PDE layer. We therefore generalize the existing `chemotaxis` and `saturation` parameters from scalars to vectors while retaining their names. Because this change generalizes the treatment of taxis toward multiple PDE fields, it is implemented in the TST framework rather than in the `ecm` model itself. For `n_chem = 1`, the TST parameter reader accepts a single value for a vector parameter, so existing one-layer parameter files remain valid: their historical scalar value becomes the single element of the vector. Here `n_chem` is the historical parameter name for the number of PDE layers.

62. Important: before making these framework changes, see [Note 13](#).

Parameter definition. In `src/parameters/parameters.hpp`, change the existing `chemotaxis` and `saturation` declarations to vector-valued parameters and require one value per PDE layer:

```
PARAMETER(std::vector<double>, chemotaxis, {0.0},
           "Taxis strength for each PDE layer")
CONSTRAINT(chemotaxis.size() == n_chem,
           "Number of chemotaxis values does not match n_chem")

PARAMETER(std::vector<double>, saturation, {0.0},
           "Saturation factor for taxis in each PDE layer")
CONSTRAINT(saturation.size() == n_chem,
           "Number of saturation values does not match n_chem")
```

63. Framework code.

Search `ca.cpp` and `deltah.cpp` for `par.chemotaxis` and `par.saturation`. Where the existing code still represents the historical one-layer behaviour, adapt the scalar accesses to `par.chemotaxis[0]` and, where applicable, `par.saturation[0]`. Recompile TST and run the one-layer vessel model again to verify that the existing chemotaxis behaviour still works with the vector-valued parameters.

64. Framework code.

Now that the existing one-layer implementation works with the vector-valued parameters, generalize the taxis calculation to all PDE layers. Add a layer-specific saturation helper to `src/cellular_potts/ca.cpp`:

```
double sat_by_layer(double c, int layer) {
    return c / (par.saturation[layer] * c + 1.0);
}
```

Then update the taxis block in `CellularPotts::DeltaH` to iterate over the PDE layers using the vector-valued `chemotaxis` and `saturation` parameters:

```
/* Chemotaxis and haptotaxis */
if (PDEfield && (par.vecadherinknockout ||
                (sxyp==0 || sxy==0))) {
    if (!(par.extensiononly && sxyp==0)) {
        for (int l=0; l<PDEfield->Layers(); l++) {
            int DDH = (int)(par.chemotaxis[l] *
                            (sat_by_layer(PDEfield->get_PDEvars(l,x,y),l) -
                             sat_by_layer(PDEfield->get_PDEvars(l,xp,yp),l)));
            DH -= DDH;
        }
    }
}
```

65. Finally, change the scalar taxis parameters to multilayer parameters in `ecm.par`:

```
chemotaxis = 6000,1000,0
saturation = 0,7,0
```

Thus the VEGF layer has unsaturated chemotaxis of strength 6000, the ECM layer has haptotaxis of strength 1000 with saturation 7, and the MMP layer does not contribute to taxis.

66. Compile and run the complete model.

In the completed simulation, endothelial cells initially occupy the vessel at the bottom of the domain and migrate through the opening toward the VEGF source. As cells encounter the ECM, haptokinesis and haptotaxis modulate their motility. VEGF stimulates MMP secretion by the cells; MMP diffuses into the surrounding tissue and locally degrades the ECM. Together with ECM-dependent proliferation, these interactions allow sprouts to extend into the domain and produce substantially more branching than in the preceding stages. The exact morphology is stochastic, so individual simulations will differ (Figure 3F). After reproducing the reference behavior, useful experiments are to vary one mechanism at a time. For example, change the VEGF chemotaxis strength, the haptokinesis or haptotaxis coefficients, the MMP secretion and degradation rates, or the proliferation parameters. Setting a mechanism to zero provides a particularly simple way to ask what role it plays in the complete model. When comparing simulations, use several random seeds because the Cellular

Potts dynamics are stochastic. Daub and Merks [22] provide examples of this one-parameter-at-a-time analysis for the original model.

This exercise illustrates a general workflow for developing models with TST. Start from an existing model that already contains the generic mechanisms you need, expose biological choices as parameters, and implement model-specific rules in the model source file. Extend the framework only when the new mechanism is genuinely reusable and is not provided by an existing interface. The model developed here deliberately uses the continuum ECM representation of the original chapter. TST also supports more advanced approaches to cell–ECM interactions; these are outside the scope of this chapter but can serve as starting points for future models.

4 Notes

- Note 1:** On some Linux distributions, Qt6 installs the qmake executable as `qmake6` rather than `qmake`. If `make` reports that `qmake` cannot be found, first check whether `qmake6` is available with `which qmake6`. If so, change `QMAKE = qmake` to `QMAKE = qmake6` in the top-level `Makefile`. On macOS with Homebrew Qt6, the executable is normally called `qmake` and no change is needed.
- Note 2:** For some parameters (*e.g.*, high diffusion coefficients) the diffusion algorithm will become numerically unstable; in those cases you can experiment with the number and size of PDE steps carried out after each Monte Carlo Step, using parameters `dt` and `pde_its`.
- Note 3:** An interesting experiment is to run the simulation in the absence of contact inhibition for these parameter values, by setting `vecadherinknockout = true`. A network-like structure may form temporarily, but it will collapse eventually.
- Note 4:** Calls of the form `PROFILE(...)` in the model source mark code regions for optional performance profiling. They do not change the biological model and are therefore not discussed further in this tutorial.
- Note 5:** Users of earlier TST versions may recognize a different organization of the PDE code. Earlier editions of this chapter implemented some model-specific source and sink terms by modifying `PDE::Secrete`, and required additional changes to the generic PDE implementation for static fields. In TST 2.0, the standard `ReactionDiffusion` path combines generic diffusion with the model-specific `PDE::DerivativesPDE` function. Spatial runtime diffusion coefficients can be set in `PDE::InitialiseDiffusionCoefficients`; setting these coefficients to zero keeps the analytical VEGF and ECM layers static without modifying `PDE::Diffuse`. New users do not need the earlier implementation.
- Note 6:** The tutorial branch can be inspected with `git diff` and abandoned without affecting the original TST branch. If you want to discard all uncommitted changes on the tutorial branch, first make sure that it contains nothing you want to keep; then restore tracked files with `git restore ..`. Untracked tutorial files can be removed separately after inspection with `git status`.

- Note 7:** Parts of the existing code are shown to give an indication of the placement of the new code. (...) marks a part of the code that is not shown.
- Note 8:** In your simulation you will now see the VEGF gradient colored in grey and green contour lines. In Figure 3A and 3B of [22] the background is white, showing the ECM concentrations, which are set to zero initially. Later on, we will add functionality to color a layer of choice.
- Note 9:** On a cluster or other machine where no interactive Qt window is required, TST automatically selects the Qt `minimal` platform when `graphics=false`, unless the user has explicitly selected another Qt platform. The platform can also be selected manually by putting the Qt option after the TST parameter-file argument, for example:

```
./ecm ../data/ecm.par -platform minimal
```

For a headless run that should still write simulation images, use `graphics=false` and `store=true`. In the TST 2.0 version used for this chapter, models render the plot before `Write()` when image storage is enabled, so stored PNG files are populated even without an interactive graphics window.

- Note 10:** Make sure to place all these parameters under `n_chem`!
- Note 11:** Until the final haptotaxis step, the tutorial uses the historical scalar `chemotaxis` and `saturation` parameters. In Section 3.5.6 these parameters are generalized to vectors without changing their names.
- Note 12:** The haptokinesis implementation used for the simulations in Ref. [22] includes contact inhibition: a haptokinesis penalty was applied maximally at cell-cell interfaces or during retraction from the ECM. The equation printed in the original article did not make this explicit. The previous edition of this chapter instead used code that followed the printed equation more literally; this produced qualitatively similar behavior after parameter adjustment, but could lead to sprout tips fanning out. In the present edition we retain the implementation used for the published simulations and state the corresponding equation explicitly. For comparison, the previous-edition code was:

```
// Haptokinesis
// Cells have optimal motility at intermediate ECM densities
int CellularPotts::Haptokinesis(int x, int y, int xp, int yp,
                                PDE * PDEfield)
{
    double s = 0.2;
    double m = 0.5;
    double c = 0.0;
    int tmpDH = 0;

    for (int l=0;l<PDEfield->Layers();l++){
        if (par.haptokinesis[l]) {
```

```

//Only during extension, look at concentration at cell site
if (!sigma[x][y])
    c = PDEfield->get_PDEvars(1,xp,yp);

double c_scaled = (c-par.min_conc[1]) / (par.max_conc[1] -
                                           par.min_conc[1]);
tmpDH += par.haptokinesis[1] *
        (-1+exp(-1*(c_scaled-m)*(c_scaled-m)/(2*s*s)))/
        sqrt(2*M_PI*s*s));
    }
}
return tmpDH;
}

```

Note 13: In later versions of TST 2.0, `chemotaxis` and `saturation` may already have been changed from scalar to vector parameters. Before carrying out these framework modifications, check `src/parameters/parameters.hpp` and verify whether they are declared as `std::vector<double>`, and search `ca.cpp` and `deltah.cpp` for `par.chemotaxis` and `par.saturation`. If these parameters are already accessed as vectors (for example, `par.chemotaxis[0]` and `par.saturation[0]`), Steps 62 and 63 can be skipped. Continue with Step 64, which generalizes the taxis calculation over PDE layers.

Table 1: Description of TST’s standard parameter file

Name in parameter file	Symbol	Default value	Use
Parameters for standard Cellular Potts model			
T	T	50	Cellular temperature or cell motility parameter
target_area	A	50	Resting area of cells
target_length	L	0	Target length of cells
lambda	λ_A	50	Strength of area constraint
lambda2	λ_L	0	Strength of length constraint
Jtable	-	J.dat	File containing values of J ; relative paths are resolved relative to the parameter file.
conn_diss	H_0	0	Energy dissipation during update that would destroy connectivity
vecadherinknockout	-	true	True if the chemotaxis algorithm is NOT contact-inhibited (see [18])
chemotaxis	χ	0	Taxis strength for each PDE layer; a single value is used by one-layer models (negative values give repulsion)
border_energy	$J_{c,B}$	100	Contact energy between cells and borders. Default value creates non-sticky boundaries. Set to make cells adhere to boundaries.
neighbours	-	3	Neighborhood order; warning: set to ‘2’ when the length constraint is used.
periodic_boundaries	-	false	If set to ‘true’ simulation uses periodic boundaries for CPM and PDE components
extensiononly	-	false	If set to ‘true’ simulation uses extension-only chemotaxis.
Parameters for partial-differential equation solver and CPM-PDE coupling			
n_chem	-	1	Number of fields in PDE component
diff_coeff	D	1e-13	Array of diffusion coefficients, typically in m^2s^{-1} . If more than one field is used, separate values with commas: e.g., 1e-13, 1e-11, 0

<code>decay_rate</code>	ϵ	1.8e-4	Array of degradation rates of chemicals, typically in s^{-1} . If more than one field is used, separate values with commas.
<code>secre_rate</code>	α	1.8e-4	Array of secretion rates of chemicals, typically in s^{-1} per lattice site. If more than one field is used, separate values with commas.
<code>saturation</code>	s	0	Saturation factor for taxis in each PDE layer; a single value is used by one-layer models
<code>dt</code>	Δt	2	Timestep of finite-difference integrator of PDE component. Typically in s.
<code>dx</code>	Δx	2.0e-6	Size of pixels for finite-difference integrator of PDE component. Typically in m.
<code>pde_its</code>	-	15	Number of finite-difference integration steps between two CPM Monte Carlo steps (MCS). I.e., integrated time between two MCS is <code>pde_its*dt</code> .
Definition of initial condition			
<code>n_init_cells</code>	-	1	Number of cells in initial condition
<code>size_init_cells</code>	-	50	Number of Eden growth [41] cycles used to generate initial cells from single pixels
<code>size_x</code>	-	200	Width of CPM and PDE field in pixels.
<code>size_y</code>	-	200	Height of CPM and PDE field in pixels.
<code>divisions</code>	-	7	Number of divisions of initial cell prior to start of simulation (used to generate a spheroid of cells).
<code>mcs</code>	-	1000000	Total number of Monte Carlo step in simulation.

rseed	-	-1	Random seed. rseed=-1 means random seed is generated from current time in milliseconds (note: do not use this option on clusters as many seeds may be generated within the same millisecond).
subfield	-	1.0	Initial cells are spread over rectangle of xsize/subfield by ysize/subfield centered in the total domain.
relaxation	-	0	Indicates the number of MCS after which the PDE component is started. This allows one to relax the CPM initial condition (e.g. allow cells to round up).
Parameters defining the simulation output			
storage_stride	-	100	Gives the number of MCS before a next image or measurement is written.
graphics	-	true	TST displays graphics on screen if set to 'true'. If set to 'false' program runs on background.
store	-	false	TST writes output to files if set to 'true' (as defined in the main simulation file).
datadir	-	data_film	Output folder for output data.

References

- [1] Oviedo, F., Kazhamiaka, F., Choukse, E., Kim, A., Luers, A., Nakagawa, M., Bianchini, R., Lavista Ferres, J.M.: Energy use of AI inference, efficiency pathways, and test-time scaling. *Joule*, 102430 (2026) <https://doi.org/10.1016/j.joule.2026.102430>
- [2] Elsworth, C., Huang, K., Patterson, D., Schneider, I., Sedivy, R., Goodman, S., Townsend, B., Ranganathan, P., Dean, J., Vahdat, A., Gomes, B., Manyika, J.: Measuring the environmental impact of delivering AI at Google Scale (2025). <https://doi.org/10.48550/arXiv.2508.15734>
- [3] Kitano, H.: Systems biology: a brief overview. *Science* **295**(5560), 1662–1664 (2002) <https://doi.org/10.1126/science.1069492>
- [4] Vergroesen, T.M., Vermeulen, V., Merks, R.M.H.: Falsifying computational models of endothelial cell network formation through quantitative comparison with

- in vitro models. *PLoS Computational Biology* **21**(4), 1012965 (2025) <https://doi.org/10.1371/journal.pcbi.1012965>
- [5] Folkman, J., Haudenschild, C.: Angiogenesis in vitro. *Nature* **288**, 551–556 (1980) <https://doi.org/10.1038/288551a0>
 - [6] Califano, J.P., Reinhart-King, C.A.: A balance of substrate mechanics and matrix chemistry regulates endothelial cell network assembly. *Cellular and Molecular Bioengineering* **1**(2–3), 122–132 (2008) <https://doi.org/10.1007/s12195-008-0022-x>
 - [7] Oster, G.F., Murray, J.D., Harris, A.K.: Mechanical aspects of mesenchymal morphogenesis. *Journal of Embryology and Experimental Morphology* **78**, 83–125 (1983)
 - [8] Manoussaki, D., Lubkin, S.R., Vernon, R.B., Murray, J.D.: A mechanical model for the formation of vascular networks in vitro. *Acta Biotheoretica* **44**(3–4), 271–282 (1996) <https://doi.org/10.1007/BF00046533>
 - [9] Manoussaki, D.: A mechanochemical model of angiogenesis and vasculogenesis. *ESAIM: Mathematical Modelling and Numerical Analysis* **37**(4), 581–599 (2003) <https://doi.org/10.1051/m2an:2003046>
 - [10] Gamba, A., Ambrosi, D., Coniglio, A., de Candia, A., Di Talia, S., Giraudo, E., Serini, G., Preziosi, L., Bussolino, F.: Percolation, morphogenesis, and Burgers dynamics in blood vessels formation. *Physical Review Letters* **90**(11), 118101 (2003) <https://doi.org/10.1103/PhysRevLett.90.118101>
 - [11] Serini, G., Ambrosi, D., Giraudo, E., Gamba, A., Preziosi, L., Bussolino, F.: Modeling the early stages of vascular network assembly. *The EMBO Journal* **22**(8), 1771–1779 (2003) <https://doi.org/10.1093/emboj/cdg176>
 - [12] Ambrosi, D., Gamba, A., Serini, G.: Cell directional persistence and chemotaxis in vascular morphogenesis. *Bulletin of Mathematical Biology* **66**(6), 1851–1873 (2004) <https://doi.org/10.1016/j.bulm.2004.04.004>
 - [13] Keller, E.F., Segel, L.A.: Initiation of slime mold aggregation viewed as an instability. *Journal of Theoretical Biology* **26**(3), 399–415 (1970) [https://doi.org/10.1016/0022-5193\(70\)90092-5](https://doi.org/10.1016/0022-5193(70)90092-5)
 - [14] Merks, R.M.H., Glazier, J.A.: A cell-centered approach to developmental biology. *Physica A: Statistical Mechanics and its Applications* **352**(1), 113–130 (2005) <https://doi.org/10.1016/j.physa.2004.12.028>
 - [15] Graner, F., Glazier, J.A.: Simulation of biological cell sorting using a two-dimensional extended Potts model. *Physical Review Letters* **69**(13), 2013–2016 (1992) <https://doi.org/10.1103/PhysRevLett.69.2013>

- [16] Merks, R.M.H., Brodsky, S.V., Goligorsky, M.S., Newman, S.A., Glazier, J.A.: Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling. *Developmental Biology* **289**(1), 44–54 (2006) <https://doi.org/10.1016/j.ydbio.2005.10.003>
- [17] Palm, M.M., Merks, R.M.H.: Vascular networks due to dynamically arrested crystalline ordering of elongated cells. *Physical Review E* **87**(1), 012725 (2013) <https://doi.org/10.1103/PhysRevE.87.012725>
- [18] Merks, R.M.H., Perryn, E.D., Shirinifard, A., Glazier, J.A.: Contact-inhibited chemotaxis in de novo and sprouting blood-vessel growth. *PLoS Computational Biology* **4**(9), 1000163 (2008) <https://doi.org/10.1371/journal.pcbi.1000163>
- [19] Szabó, A., Méhes, E., Kósa, E., Czirók, A.: Multicellular sprouting in vitro. *Biophysical Journal* **95**(6), 2702–2710 (2008) <https://doi.org/10.1529/biophysj.108.129668>
- [20] Köhn-Luque, A., de Back, W., Starrau, J., Mattiotti, A., Deutsch, A., Pérez-Pomares, J.M., Herrero, M.A.: Early embryonic vascular patterning by matrix-mediated paracrine signalling: a mathematical model study. *PLoS ONE* **6**(9), 24175 (2011) <https://doi.org/10.1371/journal.pone.0024175>
- [21] Köhn-Luque, A., de Back, W., Yamaguchi, Y., Yoshimura, K., Herrero, M.A., Miura, T.: Dynamics of VEGF matrix-retention in vascular network patterning. *Physical Biology* **10**(6), 066007 (2013) <https://doi.org/10.1088/1478-3975/10/6/066007>
- [22] Daub, J.T., Merks, R.M.H.: A cell-based model of extracellular-matrix-guided endothelial cell migration during angiogenesis. *Bulletin of Mathematical Biology* **75**(8), 1377–1399 (2013) <https://doi.org/10.1007/s11538-013-9826-5>
- [23] Scianna, M., Munaron, L., Preziosi, L.: A multiscale hybrid approach for vasculogenesis and related potential blocking therapies. *Progress in Biophysics and Molecular Biology* **106**(2), 450–462 (2011) <https://doi.org/10.1016/j.pbiomolbio.2011.01.004>
- [24] Boas, S.E.M., Merks, R.M.H.: Synergy of cell–cell repulsion and vacuolation in a computational model of lumen formation. *Journal of the Royal Society Interface* **11**(92), 20131049 (2014) <https://doi.org/10.1098/rsif.2013.1049>
- [25] Shirinifard, A., Gens, J.S., Zaitlen, B.L., Popławski, N.J., Swat, M., Glazier, J.A.: 3d multi-cell simulation of tumor growth and angiogenesis. *PLoS ONE* **4**(10), 7190 (2009) <https://doi.org/10.1371/journal.pone.0007190>
- [26] Shirinifard, A., Glazier, J.A., Swat, M., Gens, J.S., Family, F., Jiang, Y., Grossniklaus, H.E.: Adhesion failures determine the pattern of choroidal neovascularization in the eye: a computer simulation study. *PLoS Computational*

Biology **8**(5), 1002440 (2012) <https://doi.org/10.1371/journal.pcbi.1002440>

- [27] Kleinstreuer, N., Dix, D., Rountree, M., Baker, N., Sipes, N., Reif, D., Spencer, R., Knudsen, T.: A computational model predicting disruption of blood vessel development. *PLoS Computational Biology* **9**(4), 1002996 (2013) <https://doi.org/10.1371/journal.pcbi.1002996>
- [28] Abugattas-Núñez Del Prado, J., Keijzer, K.A.E., Tsingos, E., Merks, R.M.H.: Laminin and fibronectin cooperate to guide endothelial self-organization during intersegmental vessel formation. *bioRxiv* (2026) <https://doi.org/10.64898/2026.03.13.711615> . Preprint
- [29] Anderson, A.R.A., Chaplain, M.A.J.: Continuous and discrete mathematical models of tumor-induced angiogenesis. *Bulletin of Mathematical Biology* **60**(5), 857–899 (1998) <https://doi.org/10.1006/bulm.1998.0042>
- [30] Stepanova, D., Byrne, H.M., Maini, P.K., Alarcón, T.: Computational modeling of angiogenesis: the importance of cell rearrangements during vascular growth. *WIREs Mechanisms of Disease* **16**(2), 1634 (2024) <https://doi.org/10.1002/wsbm.1634>
- [31] Scianna, M., Bell, C.G., Preziosi, L.: A review of mathematical models for the formation of vascular networks. *Journal of Theoretical Biology* **333**, 174–209 (2013) <https://doi.org/10.1016/j.jtbi.2013.04.037>
- [32] Czirók, A.: Endothelial cell motility, coordination and pattern formation during vasculogenesis. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine* **5**(5), 587–602 (2013) <https://doi.org/10.1002/wsbm.1233>
- [33] Wacker, A., Gerhardt, H.: Endothelial development taking shape. *Current Opinion in Cell Biology* **23**(6), 676–685 (2011) <https://doi.org/10.1016/j.ceb.2011.10.002>
- [34] Swat, M.H., Thomas, G.L., Belmonte, J.M., Shirinifard, A., Hmeljak, D., Glazier, J.A.: Multi-scale modeling of tissues using CompuCell3D. In: *Methods in Cell Biology* vol. 110, pp. 325–366. Academic Press, ??? (2012). <https://doi.org/10.1016/B978-0-12-388403-9.00013-8>
- [35] Starrau, J., de Back, W., Brusch, L., Deutsch, A.: Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology. *Bioinformatics* **30**(9), 1331–1332 (2014) <https://doi.org/10.1093/bioinformatics/btt772>
- [36] Pitt-Francis, J., Pathmanathan, P., Bernabeu, M.O., Bordas, R., Cooper, J., Fletcher, A.G., Mirams, G.R., Murray, P., Osborne, J.M., Walter, A., Chapman, S.J., Garny, A., van Leeuwen, I.M.M., Maini, P.K., Rodríguez, B., Waters, S.L., Whiteley, J.P., Byrne, H.M., Gavaghan, D.J.: Chaste: a test-driven approach to

- software development for biological modelling. *Computer Physics Communications* **180**(12), 2452–2471 (2009) <https://doi.org/10.1016/j.cpc.2009.07.019>
- [37] Holcombe, M., Adra, S., Bicak, M., Chin, S., Coakley, S., Graham, A.I., Green, J., Greenough, C., Jackson, D., Kiran, M., Maleki-Dizaji, A., MacNeil, S., McMinn, P., Pogson, M., Poole, R., Qwarnstrom, E., Ratnieks, F., Rolfe, M.D., Smallwood, R., Sun, T., Worth, D.: Modelling complex biological systems using an agent-based approach. *Integrative Biology* **4**(1), 53–64 (2012) <https://doi.org/10.1039/C1IB00042J>
 - [38] Glazier, J.A., Graner, F.: Simulation of the differential adhesion driven rearrangement of biological cells. *Physical Review E* **47**(3), 2128–2154 (1993) <https://doi.org/10.1103/PhysRevE.47.2128>
 - [39] Steinberg, M.S.: Differential adhesion in morphogenesis: a modern view. *Current Opinion in Genetics & Development* **17**(4), 281–286 (2007) <https://doi.org/10.1016/j.gde.2007.05.002>
 - [40] Savill, N.J., Hogeweg, P.: Modelling morphogenesis: from single cells to crawling slugs. *Journal of Theoretical Biology* **184**(3), 229–235 (1997) <https://doi.org/10.1006/jtbi.1996.0237>
 - [41] Eden, M.: A two-dimensional growth process. In: *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 4, pp. 223–239. University of California Press, Berkeley, CA (1961)
 - [42] Wortel, I.M.N., Niculescu, I., Koliijn, P.M., Gov, N.S., de Boer, R.J., Textor, J.: Local actin dynamics couple speed and persistence in a cellular Potts model of cell migration. *Biophysical Journal* **120**(13), 2609–2622 (2021) <https://doi.org/10.1016/j.bpj.2021.04.036>
 - [43] Merks, R.M.H., Glazier, J.A.: Dynamic mechanisms of blood vessel growth. *Nonlinearity* **19**(1), 1–10 (2006) <https://doi.org/10.1088/0951-7715/19/1/000>
 - [44] Folkman, J.: Angiogenesis: an organizing principle for drug discovery? *Nature Reviews Drug Discovery* **6**(4), 273–286 (2007) <https://doi.org/10.1038/nrd2115>
 - [45] Pepper, M.S.: Role of the matrix metalloproteinase and plasminogen activator–plasmin systems in angiogenesis. *Arteriosclerosis, Thrombosis, and Vascular Biology* **21**(7), 1104–1117 (2001) <https://doi.org/10.1161/hq0701.093685>
 - [46] van Hinsbergh, V.W.M., Koolwijk, P.: Endothelial sprouting and angiogenesis: matrix metalloproteinases in the lead. *Cardiovascular Research* **78**(2), 203–212 (2008) <https://doi.org/10.1093/cvr/cvm102>
 - [47] Gerhardt, H., Betsholtz, C.: Endothelial–pericyte interactions in angiogenesis. *Cell and Tissue Research* **314**(1), 15–23 (2003) <https://doi.org/10.1007/>

- [48] Gerhardt, H.: VEGF and endothelial guidance in angiogenic sprouting. *Organogenesis* **4**(4), 241–246 (2008) <https://doi.org/10.4161/org.4.4.7414>
- [49] Senger, D.R., Perruzzi, C.A., Streit, M., Kotliansky, V.E., de Fougères, A.R., Detmar, M.: The $\alpha_1\beta_1$ and $\alpha_2\beta_1$ integrins provide critical support for vascular endothelial growth factor signaling, endothelial cell migration, and tumor angiogenesis. *The American Journal of Pathology* **160**(1), 195–204 (2002) [https://doi.org/10.1016/S0002-9440\(10\)64363-5](https://doi.org/10.1016/S0002-9440(10)64363-5)
- [50] Lamalice, L., Le Boeuf, F., Huot, J.: Endothelial cell migration during angiogenesis. *Circulation Research* **100**(6), 782–794 (2007) <https://doi.org/10.1161/01.RES.0000259593.07661.1e>
- [51] DiMilla, P.A., Stone, J.A., Quinn, J.A., Albelda, S.M., Lauffenburger, D.A.: Maximal migration of human smooth muscle cells on fibronectin and type IV collagen occurs at an intermediate attachment strength. *The Journal of Cell Biology* **122**(3), 729–737 (1993) <https://doi.org/10.1083/jcb.122.3.729>
- [52] Cox, E.A., Sastry, S.K., Huttenlocher, A.: Integrin-mediated adhesion regulates cell polarity and membrane protrusion through the Rho family of GTPases. *Molecular Biology of the Cell* **12**(2), 265–277 (2001) <https://doi.org/10.1091/mbc.12.2.265>
- [53] Coomber, B.L., Gotlieb, A.I.: In vitro endothelial wound repair. interaction of cell migration and proliferation. *Arteriosclerosis* **10**(2), 215–222 (1990) <https://doi.org/10.1161/01.ATV.10.2.215>
- [54] Keller, E.F., Segel, L.A.: Model for chemotaxis. *Journal of Theoretical Biology* **30**(2), 225–234 (1971) [https://doi.org/10.1016/0022-5193\(71\)90050-6](https://doi.org/10.1016/0022-5193(71)90050-6)