BREAKING BARRIERS IN MULTISCALE AGENT-BASED MODELS: EFFECTS OF CELL INDIVIDUALITY ON VIRAL INFECTION TREATMENT AND A PATH FOR CROSS-PLATFORM VALIDATION

Juliano Ferrari Gianlupi

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Luddy School of Informatics, Computing, and Engineering,

Indiana University

August 2023

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

James A. Glazier, Ph.D.

Maria Bondesson, Ph.D.

Gregory Lewis, Ph.D.

Vikram Jadhao, Ph.D.

Date of Defense: July25, 2023

To my wife, my cats, and graduate workers everywhere (in that order).

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deepest gratitude and heartfelt appreciation to each and every one of you who has supported and guided me throughout my doctoral journey. Your contributions have been invaluable, and I am immensely grateful for your presence in my life.

I am immensely grateful to my supervisor, James, for your unwavering support, expertise, and encouragement. Your guidance, intellectual insights, kind words and stern words, have been paramount in pushing me through the PhD and making my research readable and good. Your commitment to excellence have been instrumental in shaping the direction of my research and helping me grow as a researcher. I would like to extend my heartfelt appreciation to the members of my academic committee, Maria, Greg, and Vikram. Your valuable feedback, constructive criticism, and insightful suggestions have significantly enriched my research and contributed to the overall quality of my work.

To my ”in the trenches” coworkers, TJ, Josh, Priyom, and Joel, thank you for all the conversations. For helping navigate the PhD itself, the research, how to present and talk about ideas. You’ve made things infinitely better than they would have been without you.

To my fellow researchers and colleagues, both within and outside IUB, I want to thank you for sharing your knowledge, ideas, and experiences with me. The stimulating discussions, collaborative efforts, and mutual support we have shared have been instrumental in shaping my research and broadening my perspectives. Your friendship and camaraderie have made this journey all the more rewarding.

To my wife, Camila, you got me through the bad times, and are the best companion for the good times. I want to express my deepest gratitude and love. Your unwavering support, understanding, and patience have been the rock upon which I built my academic journey. You have been there for me through every triumph and challenge, providing me with boundless encouragement, and believing in me even when I doubted myself. Your

constant presence, love, and sacrifices have been the driving force behind my success. Thank you for being my pillar of strength and for standing by my side throughout this demanding journey.

To my Mom, Dad, thank you for the sacrifices you've made, the journey you took during your life. Without you I wouldn't be the person I am. I woun't have been able to do physics at UFRGS, I wouldn't have gone to TCD for a year abroad, I wouldn't have done a PhD. To my Brother, Grandparents, Aunts and Uncles, and Cousins, to my whole family, thank you for being there even from afar. Its not easy, and I miss you dearly. Your encouragement, patience, and belief in my abilities have been my constant source of motivation. I am forever grateful for your presence in my life.

I'd like to thank Rebecca Winkle for all the help with registering with ISE and all the other initial bureaucracies of grad school as well as her kindness = during the process, similarly, and not less importantly, I'd like to thank Josh Wayne for helping navigate the final bureaucracies, his pro-activeness, kindness and quick replies were fundamental to keep my anxiety in check!

I want to thank the Indiana Graduate Workers Coalition for improving working conditions for us. For fighting against such a big and powerful organization, for giving us a voice and bringing our issues to the table with IU. Thanks to us coming together as IGWC, we've done a lot. I'm thankful to have helped in the manners I could. I'm sure IGWC's members' successes are only starting!

You have all helped me grow not only as a researcher but also as an individual.

Finally, I also want to thank Liang Chen for creating the LaTeX template I used to write this thesis (available at `https://github.com/liang-chen/IU-PhD-Thesis-Template`), as well as Leslie Lamport and Donald Knuth for originating LaTeXitself. Without their efforts writing a thesis would be impossible.

Juliano Ferrari Gianlupi

BREAKING BARRIERS IN MULTISCALE AGENT-BASED MODELS: EFFECTS OF CELL INDIVIDUALITY ON VIRAL INFECTION TREATMENT AND A PATH FOR CROSS-PLATFORM VALIDATION

This Ph.D. research focuses on two primary areas: investigating human health questions using mechanistic Agent-based models (ABMs) of cells and tissues, and enhancing the field of mechanistic bio-ABM by improving model verification and sharing. ABMs offer a bottom-up approach to studying complex biological systems by capturing individual-level behaviors and interactions. The thesis specifically concentrates on agent-based models of cell tissues, with a particular emphasis on COVID-19 and anti-viral treatment. The research explores the integration of traditional pharmacokinetics-pharmacodynamics (PKPD) or physiologically based pharmacokinetics (PBPK) models with ABMs to gain insights COVID-19 treatment with remdesivir. It investigates factors contributing to the low adoption of remdesivir as a COVID-19 treatment, such as the harsh treatment schedule and potency miss-estimation, with a focus on cell response heterogeneity as a potential cause for miss-estimation. Additionally, the research highlights the importance of model cross-platform portability. Running the same model in different platforms can validate models, and ensure model robustness. However, porting a model is difficult and time consuming, emphasizing the need for a universal modeling description standard for ABMs akin to the Systems Biology Markup Language (SBML). To investigate some of the challenges of developing such a standard, this research creates a model specification translation to translate a PhysiCell model into a CompuCell3D model.

James A. Glazier, Ph.D.

Maria Bondesson, Ph.D.

Gregory Lewis, Ph.D.

Vikram Jadhao, Ph.D.

TABLE OF CONTENTS

Acknowledgements v

Abstract vii

List of Tables xvi

List of Figures xvii

Chapter 1: Introduction and Background 1

1.1 Modeling SARS-CoV-2 Infection and Treatment 3

1.2 Cross-Platform Validation 4

Chapter 2: An overview of the Sego-Aponte-Gianlupi COVID-19 model 6

2.1 Introduction 6

2.2 Overview of the Sego-Aponte-Gianlupi Model 7

2.2.1 Epithelial cells in the Sego-Aponte-Gianlupi model 9

2.2.2 Diffusion of Virus, Cytokines, and Oxidizing Chemical in the Sego-Aponte-Gianlupi Model 10

2.2.3 Viral Life-Cycle Model 11

2.2.4 Immune system model 13

2.2.5 Preliminary Antiviral treatment 14

2.3 Sego-Aponte-Gianlupi Selected Results 15

2.4 Sego-Aponte-Gianlupi Discussion 18

Chapter 3: Multiscale Model of Antiviral Timing, Potency, and Heterogeneity Effects on an Epithelial Tissue Patch Infected by SARS-CoV-2 . . . 19

3.1 Introduction 19

3.2 Materials and Methods 24

3.2.1 Changes to the Sego-Aponte-Gianlupi model 24

3.2.2 Remdesivir physiologically based pharmacokinetic model 25

3.2.3 Remdesivir mode of action (MOA) model 27

3.2.4 Heterogeneous cellular metabolism of remdesivir modeling 28

3.2.5 Simulating antiviral treatment regimens and treatment classification metrics 30

3.3 Results 33

3.3.1 Remdesivir PK model 33

3.3.2 Variability of outcomes in Sego-Aponte-Gianlupi 's model 34

3.3.3 Predictive treatment outcomes 38

3.4 Discussion 56

Chapter 4: Translating model specifications 61

4.1 Introduction 61

4.2 Conceptual model differences between CompuCell3D and PhysiCell models 64

4.3 An Overview of the Dynamics of CompuCell3D and PhysiCell 66

4.3.1 Dynamics in CompuCell3D 67

4.3.2 Dynamics in PhysiCell 71

4.4 Implementation of the Translation Process 74
4.4.1 Translating Space 74
4.4.2 Translating Time 76
4.4.3 Extracting Cell Types and Constraints 78
4.4.4 Extracting Diffusing Elements 79
4.4.5 Secretion and Uptake 81
4.5 Challenges 82
4.5.1 Appropriate Cell and Simulation Domain Sizes 83
4.5.2 Appropriate Diffusion Parameters 89
4.5.3 Cell-Cell Adhesion & Repulsion Implementation 90
4.5.4 CPM Limitation on Cell Speed 91
4.6 Generating the CompuCell3D Simulation 91
4.6.1 Creating the XML file 92
4.6.2 Creating Steppables File 101
4.7 Results (example translations) 112
4.7.1 Cell Cycle 112
4.7.2 Biorobots 113
4.8 Discussion 116
4.8.1 Limitations & Future Work 117
4.8.2 Software requirements 119

Chapter 5: PhenoCellPy: A Python package for biological cell behavior modeling 120

5.1 Introduction 120

5.2 PhenoCellPy Overview . . . 124
5.2.1 Cell Volume Class . . . 125
5.2.2 Phase Class . . . 127
5.2.3 Phenotype Class . . . 131
5.2.4 Using PhenoCellPy . . . 131
5.3 Pre-defined Phenotypes . . . 137
5.3.1 Simple Live Cycle . . . 137
5.3.2 Ki-67 Basic . . . 137
5.3.3 Ki-67 Advanced . . . 137
5.3.4 Flow Cytometry Basic . . . 138
5.3.5 Flow Cytometry Advanced . . . 138
5.3.6 Apoptosis Standard . . . 138
5.3.7 Necrosis Standard . . . 139
5.4 Selected Examples . . . 140
5.4.1 CompuCell3D . . . 140
5.4.2 Tissue Forge . . . 141
5.5 Selected Results . . . 144
5.5.1 CompuCell3D . . . 144
5.5.2 Tissue Forge . . . 150
5.6 Discussion . . . 152
5.6.1 Installation . . . 152
5.6.2 Planned features . . . 152
5.6.3 Requirements . . . 153

Chapter 6: Discussion 154

6.1 Other Infrastructure Work 155

6.2 Future work 156

6.2.1 Translator 156

6.2.2 PhenoCellPy 157

6.2.3 My Future Post-Doctoral Research 157

Appendix A: Multiscale Model of Antiviral Timing, Potency, and Heterogeneity Effects on an Epithelial Tissue Patch Infected by SARS-CoV-2 158

A.1 Simple PK model for Remdesivir and GS-443902 158

A.1.1 COPASI Codes 161

A.2 Table of parameters from Sego *et. al* 163

A.3 Quantitative metrics of treatment outcome 166

A.4 Instructions for running the multiscale CompuCell3D simulations and for analyzing the results 167

A.4.1 CompuCell3D simulations 167

A.4.2 Results analysis 169

A.5 Supplementary results for the untreated simulations with different initial conditions 169

A.6 Supplementary results from treatment initiation delay, antiviral potency, and GS-443902 half-life variation 172

A.6.1 Homogeneous metabolism, regular GS–443902 half-life 173

A.6.2 Homogeneous metabolism, halved GS–443902 half-life 190

A.6.3 Homogeneous metabolism, GS–443902 half-life reduced by 75% . 204

A.6.4 Heterogeneous metabolism, regular GS–443902 half-life 209

A.6.5 Heterogeneous metabolism using other standard deviations 226

A.7 Supplementary results for viral production metabolism rate correlation . . . 233

Appendix B: Translating Model Specifications Appendix 238

B.1 Converting Space Dimensions 238

B.2 Converting Time Dimensions 241

B.3 Converting Cell Types and Extracting Mechanics 241

B.3.1 Cell Types Extraction 241

B.3.2 Mechanics Extraction 243

B.4 Converting Secretion and Uptake rates 247

B.5 Re-scaling time 249

B.6 Initial conditions 251

B.7 Generating diffusion's XML 254

B.8 Stepabble generation helper functions 274

B.9 Generate constraint loops helper function 277

B.10 Generating phenotype models initialization 279

Appendix C: PhenoCellPy Appendix 284

C.1 PhenoCellPy Python implementation 284

C.1.1 Cellular Phase 284

C.1.2 Cellular Phenotype 288

C.1.3 Cellular Volume 294

C.2 Phase init function 297

C.3 Phase init function 301

C.4 Ki-67 Basic Cycle Improved Division Implementation in CompuCell3D . . 304

C.5 Ki-67 Basic Cycle Implementation in Tissue Forge 309

References 312

Curriculum Vitae

LIST OF TABLES

3.1 List of parameters used for the minimal PBPK model of remdesivir. 27

3.2 List of parameters for the ABM and PD models as well as parameters varied for the treatment effectiveness investigation 30

4.1 Comparison of selected concepts from CompuCell3D and PhysiCell 66

5.1 Volumes and volume change rates defined by the Cell Volume class. Volumes marked with * are the dynamic volumes. 127

A.1 Data used to calibrate the simple remdesivir to GS–443902 prediction model. *Humeniuk's Table 4 in [23]; **EU Compassionate Use's Table 16 in [112]; *** Infusion duration not given, assumed to be 1 hour. 160

A.2 Sego et al.'s conversion factors 163

A.3 Sego et al.'s parameters part 1 164

A.4 Sego et al.'s parameters part 2 165

A.5 Sego et al.'s parameters part 3 165

LIST OF FIGURES

2.1 Results of one of the simulations from the antiviral investigation paper [2]. Top row is the epithelial 2D layer, epithelial cells can be uninfected (blue), infected in the eclipse phase (green), infected secreting virus (red), or dead (black). Middle row shows the virus concentration field. Botom row is the immune cells (burgundy) 2D layer. Adapted from [2]. 7

2.2 Simulation time-courses. A) Population of epithelial cells in different infection stages, uninfected in orange, infected eclipse phase in green, infected releasing virus in red, and dead cells in purple. B) Total diffusing cytokine and virus (arbitrary units), cytokine in magenta, and virus in brown. C) Immune model data, simulated domain immune cell population (grey), immune recruitment signal (S, Equation 2.12). Adapted from [1]. 16

2.3 sensitivity analysis on the viral binding affinity constant (k_{on}), and the immune system delay constant (β_{delay}). 2.3a) . Adapted from [1]. 17

3.1 Schematic diagram of the minimalized PBPK model of remdesivir. PBMCs are a surrogate for lung alveolar epithelial cells for GS–443902. 26

3.2 Simulated epithelial cell layers colored by the change in k_{in} 3.2a, k_{out} 3.2b. The values displayed are relative to the base k_{in} and k_{out}, a cell colored blue in 3.2b has, *e.g.*, $k'_{out}(\sigma) = 0.3 \times k_{out}$ 29

3.3 3.3a Concentration of GS–443902 (remdesivir's active metabolite) for a 14 days treatment with a 200 mg loading dose and 100 mg subsequent daily doses (IV infusion) is obtained by solving Equations 3.1a and 3.1a. Concentration peaks (red) and troughs (blue) are pointed out, their midpoint (dashed line) is our base IC_{50}. 3.3b Concentrations of the active metabolite, GS–443902, in PK simulations for the different dosing regimens of remdesivir, the doses are rescaled to keep the total average amount of remdesivir given over 24 hours constant. 3.3c Some selected PK profiles from 3.3b. 34

3.4 Uninfected cell populations for 400 replicas of the Sego-Aponte-Gianlupi model [1] are shown using Sego-Aponte-Gianlupi 's default parameters [1]. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. 3.4a) Simulations start with 1 initially infected cell and 63 simulations result in "*failure to infect*" (15.75% of replicas), the 90th quantile includes the upper bound of the number of cells. 3.4b) Simulations start with 2 initially infected cells where 5 simulations result in "*failure to infect*" (1.25%), the 100th quantile includes the upper bound of the number of cells. 3.4c) Simulations start with 5 initially infected cells. 3.4d) Simulations start with 10 initially infected cells. 36

3.5 Extracellular viral load curve for untreated simulations with five initially infected cells in the tissue patch. 38

3.6 Coarse parameter investigation (10 replicas of the treatment simulation). Treatment starts with 10 infected cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Treatments with an IC_{50} multiplier < 0.055 contain the infection, while treatments with IC_{50} multiplier ≥ 0.05 do not. The top two rows show a reduction of viral load due to treatment, while in the lower two rows the decrease is due to all cells being dead 3.6a Uninfected cell population. 3.6b Extracellular diffusive virus; *y*-axis in log scale, exponent values as tick-marks. 40

3.7 Treatment starts with the infection of 10 epithelial cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.7a Uninfected cell population. 3.7b Extracellular diffusive virus; *y* axis in log scale, exponent values as tick-marks. 42

3.8 Treatment starts three days post the infection of 10 epithelial cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.8a Uninfected cell population. 3.8b Extracellular diffusive virus; *y*-axis in log scale, exponent values as tick-marks. 44

3.9 Replica snapshots of the tissue patch are shown for different treatment classifications. In all the cases the top row is the epithelial layer (blue uninfected cells, green infected cells in eclipse phase, red infected cells releasing virus, black dead cells), middle row is the extracellular virus concentrations (high concentration in red, low concentration in blue), and the third row is the immune cell layer (immune cells in red, extracellular environment in black). **A)** fast clearance (36h dosing period, 0.01 IC_{50} multiplier), **B)** slow clearance (84h dosing period, 0.05 IC_{50} multiplier), **C)** partial containment (84h dosing period, 0.06 IC_{50} multiplier), **D)** widespread infection (108h dosing period, 0.07 IC_{50} multiplier). In all the cases snapshots are shown at the start of the simulation (day 0), at the start of treatment (3 days post infection of 10 cells), after 14 days of treatment, and at the end of the simulation (Day 28). 45

3.10 Extracellular diffusive virus populations for 8 replicas of the treatment simulation. Treatment starts with the infection of 10 cells, the half-life of GS–443902 was halved (to 15.2h). For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 46

3.11 Extracellular diffusive virus populations for eight replicas of the treatment simulation. Epithelial cells' metabolism and clearance varies from cell to cell. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.11a Treatment starts with the infection of 10 cells. 3.11b Treatment starts three days post the infection of 10 cells. 48

3.12 Mean viral production of cells versus their relative metabolic rates normalized by the maximum mean production with partial containment parameters. 3.12a Results for simulations varying only k_{in}, uses the partial containment parameter set. 3.12b Results for simulations varying only k_{out}, uses the widespread infection parameter set. 50

3.13 Extracellular diffusive virus populations for eight replicas of the treatment simulation. Epithelial cells' metabolism and clearance varies from cell to cell. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.13a Cells' metabolism standard deviation set to 0.1. 3.13b Cells' metabolism standard deviation set to 0.5. 52

3.14 Ineffective-effective treatment transition border for different levels of metabolic variability. In blue is the homogeneous metabolism case, in black the heterogeneous metabolism case applying a modulation by a Gaussian random number with standard deviation (S.D.) set to 0.1 for k_{in} and k_{out}, in red the heterogeneous case with the Gaussian random number S.D. set to 0.25, in green with the S.D. set to 0.5. 53

3.15 Mean RNA levels in virus-releasing infected cells (solid lines) with standard deviation (shaded regions) versus time for individual simulation replicates under different simulation options. Sub-figure 3.15a is untreated. Sub-figure 3.15b is treated with no metabolic heterogeneity, 3.15c is treated with a metabolism standard deviation of 0.1, 3.15d is treated with a metabolism standard deviation of 0.25, and 3.15e is treated with a metabolism standard deviation of 0.5. Please note that the y-range in 3.15a and 3.15b differ from the others. 55

4.1 Example GUI of a running simulation in: 4.1a) PhysiCell and 4.1b) CompuCell3D. Both figures show the a simulation that was translated using the methods described here: biorobots, see Section 4.7.2. 64

4.2 Pixel neighborhoods for each order, main pixel in red, neighborhood in blue. 4.2a) 1st order, 4.2b) 2nd order, 4.2c) 3rd order, 4.2d) 4th order. . . . 69

4.3 Translated cell cycle simulation. 4.3a) The cells in the simulation. 4.3b) Color coded cells based on which phase of the cycle they are in. 113

4.4 Translated biorobots simulation. 4.4a) The color-coded cells (cargo in blue, workers in red, and directors in yellow). 4.4b) The cargo cells' chemo-attractant field levels. 4.4c) The directors' chemo-attractant field levels. 114

5.1 Example sequences of cell behaviors. a) Cell cycle. b) Stages of viral infection of a cell. 121

5.2 How PhenoCellPy is organized.Gray boxes are PhenoCellPy classes, the blue trapezoid is the lists of constituent Phases. Ownership of objects is conveyed through overlaying shapes (*e.g.*, the Phenotype owns the list of Phases, the Phase owns the Cell Volume). Yellow arrows indicate sequence in the list of Phases. Any Phase could go to any other Phase, as dictated by the modeler. A Phase can have exits to any number of Phases, and a Phase can have entrances from multiple Phases. 125

5.3 Cell Volume class attributes and functions (5.3a), and Cell Volume update volume (5.3b). The Cell Volume update is highlighted by the black outline. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, green diamonds are decisions. Yellow arrows mean function call, blue arrows are information being passed. 128

5.4 Phase class attributes and functions (5.4a), and Phase time-step flowchart (5.4b). The time-step function is highlighted by the black outline, the order in which it performs operations is overlayed on the arrows. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, yellow diamonds are decision-making functions. Yellow arrows mean function call, blue arrows are information being passed. 130

5.5 Phenotype class attributes and functions (5.5a), and Phenotype time-step flowchart (5.5b). The time-step function is highlighted by the black outline, the order in which it performs operations is overlayed on the arrows. Purple hexagon represents the model PhenoCellPy is embedded in (*i.e.*, the main model), the gray pentagon is an agent from the main model. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, green diamonds are decisions. Yellow arrows mean function call, blue arrows are information being passed. 132

5.6 Spatial figures for the CompuCell3D simulation using the regular Ki-67 Basic cycle. a, b, and c) shows the cell cluster with no color overlay. They show the cluster at the start of the simulation, step 2000 and step 4000 respectively. d) Zoom in on the center of image c. e) Color coded cells based on the phase they are in, green for the quiescent phase and red for the proliferating phase. Step 2800. 144

5.7 Statistics for the cell population in CompuCell3D using the standard Ki-67 Basic cycle. a) Total cell population. b) Cell population volume statistics. Maximum cell volume in red, median in yellow, and minimum in blue. 146

5.8 Spatial figures for the CompuCell3D simulation using the regular Ki-67 Basic cycle. a, b, and c) shows the cell cluster with no color overlay. They show the cluster at the start of the simulation, step 2000 and step 4000 respectively. d) Zoom in on the center of image c. e) Color coded cells based on the phase they are in, green for the quiescent phase and red for the proliferating phase. Step 3200. 147

5.9 Statistics for the cell population in CompuCell3D using the modified Ki-67 Basic cycle. a) Total cell population. b) Cell population volume statistics. Maximum cell volume in red, median in yellow, and minimum in blue. 148

5.10 Snapshots of the CompuCell3D simulation for the necrotic phenotype, necrotic cells in green, healthy cells in blue, fragmented cells in red. a) simulation start. b) Necrotic cells at maximum volume. c) Moment the necrotic cells bursts. d) end of the simulation. 149

5.11 Necrotic cells volumes evolution in time. Each necrotic cell volume is plotted individuality 150

5.12 Cells space configuration in the Tissue Forge Ki-67 Basic cycle model. a) Simulation start. b) Day 7. c) Day 15 (simulation end). 151

5.13 Cell population statistics for the Tissue Forge model using PhenoCellPy's Ki-67 Basic Cycle phenotype. a) Total cell population. b) Cells' volume statistic. 151

A.1 Comparison of our simplified model to Gallo's population simulation plot of predicted intracellular lung GS–443902 concentration. Gallo's mean response is shown by the black line and the 95% interval is shaded. This data is from 5000 simulations in which the two key parameters in the Gallo model were sampled from distributions with 20% CV. See Gallo and their supplement Figure S5 for more information. Blue line is our simplified model's output. The Gallo data used in this plot was digitized from the publication using `https://automeris.io/WebPlotDigitizer/`. 161

A.2 Dead cell populations for 400 replicas of Sego *et al.*'s model [1]. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.2a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.2b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.2c) Simulations start with 5 initially infected cells. A.2d) Simulations start with 10 initially infected cells. . . . 170

A.3 Extracellular viral load for 400 replicas of Sego *et al.*'s model [1], y-axis in log scale. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.3a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.3b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.3c) Simulations start with 5 initially infected cells. A.3d) Simulations start with 10 initially infected cells. . . . 171

A.4 Extracellular viral AUC for 400 replicas of Sego *et al.*'s model [1], y-axis in log scale. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.4a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.4b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.4c) Simulations start with 5 initially infected cells. A.4d) Simulations start with 10 initially infected cells. . . . 172

A.5 Uninfected population. . . . 173

A.6 Infected (eclipse phase) population. . . . 173

A.7 Infected (secreting extracellular virus) population. . . . 174

A.8 Dead population. . . . 174

A.9 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. . . . 175

A.10 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 175

A.11 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 176

A.12 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 176

A.13 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 177

A.14 Uninfected population. 177

A.15 Infected (eclipse phase) population. 178

A.16 Infected (secreting extracellular virus) population. 178

A.17 Dead population. 179

A.18 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 179

A.19 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 180

A.20 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 180

A.21 Uninfected population. 181

A.22 Infected (eclipse phase) population. 181

A.23 Infected (secreting extracellular virus) population. 182

A.24 Dead population. 182

A.25 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 183

A.26 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 183

A.27 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 184

A.28 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 184

A.29 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 185

A.30 Uninfected population. 185

A.31 Infected (eclipse phase) population. 186

A.32 Infected (secreting extracellular virus) population. 186

A.33 Dead population. 187

A.34 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 187

A.35 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 188

A.36 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 188

A.37 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 189

A.38 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 189

A.39 Uninfected population. 190

A.40 Infected (eclipse phase) population. 190

A.41 Infected (secreting extracellular virus) population. 191

A.42 Dead population. 191

A.43 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 192

A.44 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 192

A.45 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 193

A.46 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 193

A.47 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 194

A.48 Uninfected population 194

A.49 Infected (eclipse phase) population. 195

A.50 Infected (secreting extracellular virus). 195

A.51 Dead population. 196

A.52 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 196

A.53 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 197

A.54 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 197

A.55 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 198

A.56 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 198

A.57 Uninfected population. 199

A.58 Infected (eclipse phase) population. 199

A.59 Infected (secreting extracellular virus) population. 200

A.60 Dead population. 200

A.61 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 201

A.62 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 201

A.63 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 202

A.64 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 202

A.65 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 203

A.66 Uninfected population. 204

A.67 Infected (eclipse phase) population. 204

A.68 Infected (secreting extracellular virus) population. 205

A.69 Dead population. 205

A.70 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 206

A.71 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 206

A.72 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 207

A.73 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 207

A.74 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 208

A.75 Uninfected population. 209

A.76 Infected (eclipse phase) population. 209

A.77 Infected (secreting extracellular virus) population. 210

A.78 Dead population. 210

A.79 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 211

A.80 Total diffusive virus produced (AUC) for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 211

A.81 Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 212

A.82 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 212

A.83 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 213

A.84 Uninfected population. 213

A.85 Infected (eclipse phase) population. 214

A.86 Infected (secreting extracellular virus) population. 214

A.87 Dead population. 215

A.88 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 215

A.89 Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 216

A.90 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 216

A.91 Uninfected population. 217

A.92 Infected (eclipse phase) population. 217

A.93 Infected (secreting extracellular virus) population. 218

A.94 Dead population. 218

A.95 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 219

A.96 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 219

A.97 Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 220

A.98 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 220

A.99 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 221

A.100 Uninfected population. 221

A.101 Infected (eclipse phase) population. 222

A.102 Infected (secreting extracellular virus) population. 222

A.103 Dead population. 223

A.104 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 223

A.105 Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks. 224

A.106 Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks. 224

A.107 Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks. 225

A.108 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 225

A.109 Uninfected population. 226

A.110 Infected (eclipse phase) population. 226

A.111 Infected (secreting extracellular virus) population. 227

A.112 Dead population. 227

A.113 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 228

A.114 Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 228

A.115 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 229

A.116 Uninfected population. 229

A.117 Infected (eclipse phase) population. 230

A.118 Infected (secreting extracellular virus) population. 230

A.119 Dead population. 231

A.120 Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks. 231

A.121 Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks. 232

A.122 Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state. 232

A.123 Mean antiviral drug levels in virus-releasing infected cells (solid lines) with standard deviation (shaded regions) versus time for individual simulation replicates under different simulation options. Sub-figure A.123a is treated with no metabolism heterogeneity, A.123b is treated with a metabolism standard deviation of 0.1, A.123c is treated with a metabolism standard deviation of 0.25, and A.123d is treated with a metabolism standard deviation of 0.25. 233

A.124 Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with rapid clearance parameters. A.124a Results for simulations varying only k_{in}. A.124b Results for simulations varying only k_{out}. 234

A.125 Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with rapid clearance parameters. A.125a Results for simulations varying only k_{in}. A.125b Results for simulations varying only k_{out}. 235

A.126 Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with slow clearance parameters. A.126a Results for simulations varying only k_{in}. A.126b Results for simulations varying only k_{out}. 236

A.127 Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with widespread infection parameters. A.127a Results for simulations varying only k_{in}. A.127b Results for simulations varying only k_{out}. 237

CHAPTER 1
INTRODUCTION AND BACKGROUND

My Ph.D. research has two areas of focus. The first is investigating questions related to human health using mechanistic Agent-based models (ABMs) of cells and tissues. The second is strengthening the field of mechanistic bio-ABM by investigating how to make models easier to be verified and shared.

Agent-based models are powerful tools for investigating complex biological systems [1–9], offering a bottom-up approach that captures individual-level behaviors and interactions. This thesis aims to contribute to the advancement of mechanistic modeling in human health by focusing on agent-based models of cell tissues, with a particular emphasis on COVID-19 and anti-viral treatment for it. In Chapters 2 and 3 I present my work on COVID-19 models (published in [1] and [2]).

My research explores the dynamics of COVID-19 within a patch of the lung. It investigates how can more traditional pharmacokinetics-pharmacodynamics (PKPD) or physiologically based pharmacokinetics (PBPK) models can be integrated with ABMs. A key objective is to examine the insights that can be gained from the combination of traditional pharmacometrics models and ABMs, particularly in relation to the low adoption of remdesivir as a COVID-19 treatment. The research investigates potential factors for the low adoption: such as the harsh treatment schedule and drug potency miss-estimation. It identifies response heterogeneity of cells as a possible cause for the potency miss-estimation. Probing response heterogeneity by cells can't be done with more traditional PBPK.

In science, our experiments (models, simulations, in the case of computational science) should be replicatable and reproducible using a different method, which provides a means of comparing results and ensuring model robustness. Tissue agent-based models lack these qualities, causing a crisis of reproducibility. Reproduction can be done if the model is

re-implemented by hand in a different platform. However, the process of cross-platform validation is time-consuming and challenging, requiring the translation and adaptation of model specifications from one framework to another. Besides being an imperfect application of the scientific method, the lack of these qualities hinders collaboration and causes a reproducibility crisis.

My work encompasses the creation of a method to bridge the replication gap between two modeling platforms: CompuCell3D [10] and PhysiCell [11]. I created a translator to go from the model specification of one platform to the specification of another, see Chapters 4 (unpublished). In the future, the field of tissue ABMs can be made stronger by the creation of a universal modeling description standard akin to the Systems Biology Markup Language (SBML) [12] for ABMs. My translator is a step in the direction of the universal model spec, as it investigates possible pitfalls and find solutions to som eof them.

Another issue with replication is the miss-match of concepts. If platform A has a model of, *e.g.*, cell shape and platform B does not, we need to either make the cell shape model available to platform B, or decide what to do about the missing concept. In Chapter 5 (published in [13]) I present my work on making PhysiCell's cell phenotype sub-models available to any Python-based modeling platform by re-imagining, and re-implementing, them in pure Python.

Besides the work presented in this thesis, I have established methods to compare population dynamics, ordinary differential equation (ODE), models with ABMs, and how to transform an ODE model into an ABM [3]. I have ongoing work investigating interferon pathways in cellular innate immune response, and ongoing work into the replication of leishmania in the infection site and immune recruitment to the infection site. I have also published several online educational tools to demonstrate CompuCell3D's capabilities.

1.1 Modeling SARS-CoV-2 Infection and Treatment

The COVID-19 pandemic has underscored the need for comprehensive modeling approaches to better understand the disease dynamics, transmission patterns, and the effectiveness of interventions. Agent-based models, capable of capturing the spatial and temporal aspects of cell behavior, offer a unique opportunity to simulate the complex interactions between immune cells, viral particles, and therapeutic agents within tissues. By incorporating detailed biological mechanisms and experimental data, these models have the potential to provide valuable insights into the underlying processes of COVID-19 and aid in the development of targeted treatment strategies.

In this thesis I explain our first SARS-COV-2 model [1] (see Chapter 2). We aimed to replicate the reported infection time-line and multiple infection outcomes. We investigated the interplay of viral infectivity and immune response intensity to question if those two parameters are enough to replicate the wide-array of outcomes. We only had palliative treatments available at the time, some people recovered or died quickly, while some were sick for weeks. We also did a preliminary implementation of antiviral treatment, that I expanded in [2] (see Chapter 3).

In my COVID-19 remdesivir treatment work [2] (see Chapter 3), I investigate how to integrate a physiologically based pharmacokinetics (*PBPK*) and pharmacodynamics (*PD*) model with an ABM, and what other questions can this pairing ask. I investigated what are the effects of cell individuality on the overall treatment, this is a pertinent question because remdesivir is a prodrug that has to be metabolized intracellularlly into its active form [14].

The COVID-19 pandemic has highlighted the urgent need for efficient model development and collaborative efforts among research groups to gain valuable insights into the complex dynamics of the disease. Furthermore, the ability to compare and validate models is crucial for ensuring their reliability and applicability. A promising approach to model validation involves implementing the same underlying biological processes in dif-

ferent computational frameworks. However, the process of re-implementing a model in a new platform is often challenging and time-consuming, posing a significant barrier to cross-platform validation. My thesis begins to address this difficulty.

1.2 Cross-Platform Validation

Agent-based modeling is a powerful tool for studying complex biological systems, offering a unique perspective on the dynamics and interactions of individual entities within a population. This approach has shown great promise for the investigation of the mechanics of tissue behaviors, from cancer [5, 6] to embryonic development [7–9] and pathogen infections [1–4]. They enable the investigation of intricate cellular behaviors and their impact on disease progression, treatment strategies, and therapeutic interventions.

One crucial aspect of advancing agent-based modeling in human health is the cross-platform validation of models. Model validation plays a pivotal role in ensuring the reliability and predictive accuracy of simulations. However, due to the diversity of modeling frameworks and methodologies, validating agent-based models across different platforms presents significant challenges. This thesis addresses this issue by proposing a prototype method for cross-platform validation of agent-based models of cell tissues, bridging the gap between various modeling paradigms.

Cellular Potts models (CPMs) and center-based models (CBMs) are two commonly employed approaches in agent-based modeling of cell tissues, each with its strengths and limitations. There are two possible strategies for bridging the gap: creating a general model description, or by translating model specification from one platform to another. In this thesis I explored the translation of PhysiCell into CompuCell3D, exploring the necessary adaptations and overcoming the conceptual and computational differences between the methodologies. By enabling the conversion of models from one methodology to another, this research aims to enhance the accessibility and applicability of agent-based models for studying human health, see Chapter 4. I have also implemented PhysiCell's phenotype

submodel [11] into an independent Python package that can be used by any Python-based modeling platform (see Chapter 5).

CHAPTER 2

AN OVERVIEW OF THE SEGO-APONTE-GIANLUPI COVID-19 MODEL

In this chapter I will present the work we did with the Sego-Aponte-Gianlupi model [1]. This was a joint project with several people in my lab, I then iterated on this model to create my remdesivir model [2] (see Chapter 3). My colleague Joshua Aponte presented most of this work in his dissertation, therefore I will present the most relevant parts of the Sego-Aponte-Gianlupi model to my Remdesivir model in this Chapter. This Chapter was creating by adapting a section of the Remdesivir model publication [2].

It was very clear at the start of pandemic that we needed a standard to share our methods and models with other teams, and ways to make sure our models can be easily iterated on and expanded. This lead to my investigations on how to implement models cross-platform (see Chapters 4 and 5).

We began this work early in the pandemic, the goals of this work were to make a simulation that replicated the viral time course in a human patient, the variability of outcomes in a human patient that hasn't been infected before, and laying the groundwork for an extensible and reusable framework for multiscale models of infection. We also used this model to do preliminary investigations of possible treatments. I later carried on the antiviral treatment investigation in [2], see Chapter 3.

2.1 Introduction

To model the spread of viral infection, cytokine response, and immune cell activity in a tissue, we created a hybrid multiscale agent-based model of SARS-CoV-2 infection in a small lung tissue patch, the Sego-Aponte-Gianlupi model [1]. In this model, the infection progresses as follows: the initially infected cell(s) goes through the viral life cycle; the infected cell(s) could die during the eclipse phase, if it survives it releases virus into the

extracellular space (after the eclipse phase) and continues to release virus until it dies. The virus diffuses and decays extracellularly according to Fick's law (see Section 2.2), The description of the Sego-Aponte-Gianlupi model is extensive [1], Section 2.2. presents a summarized description of the Sego-Aponte-Gianlupi model. It should be noted that with the default parameters used in the Sego-Aponte-Gianlupi model all the simulated epithelial cells are eventually infected and die [1]. However, depending on the parameters used, we also saw containment of the infection by the immune system [1].

2.2 Overview of the Sego-Aponte-Gianlupi Model

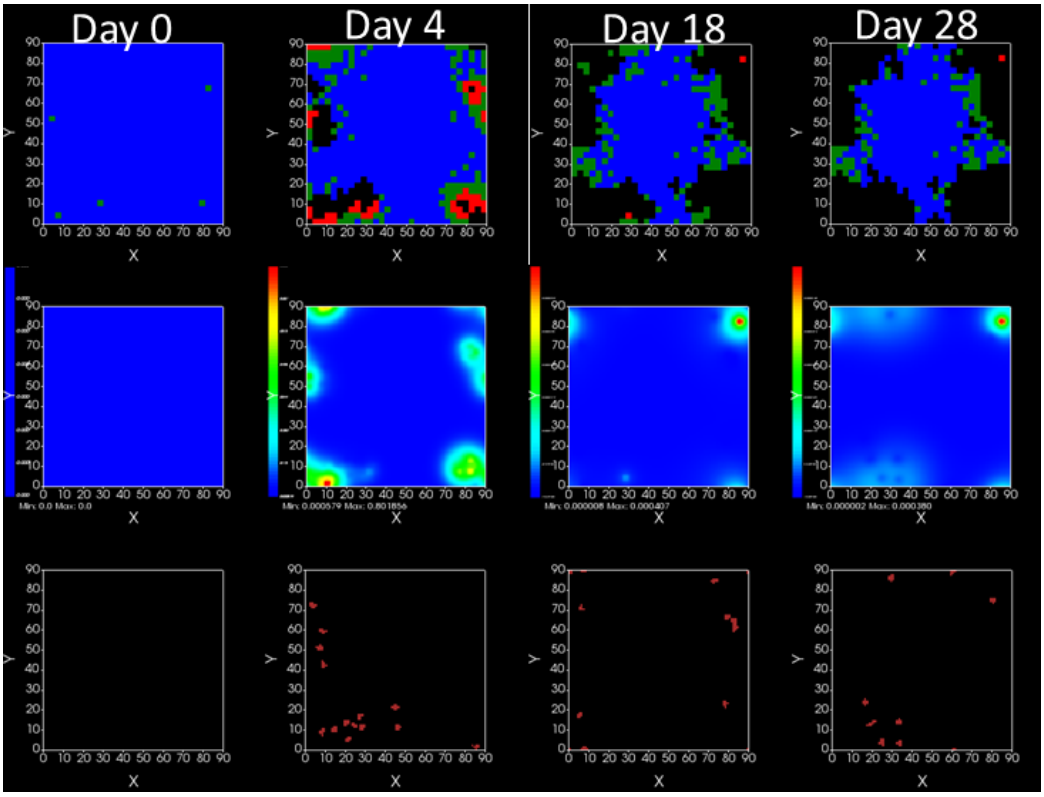


Figure 2.1: Results of one of the simulations from the antiviral investigation paper [2]. Top row is the epithelial 2D layer, epithelial cells can be uninfected (blue), infected in the eclipse phase (green), infected secreting virus (red), or dead (black). Middle row shows the virus concentration field. Botom row is the immune cells (burgundy) 2D layer. Adapted from [2].

The Sego-Aponte-Gianlupi [1] agent-based model is a 2.5-dimensional Cellular Potts model (*CPM*), and is implemented in CompuCell3D [10]. We use CompuCell3D version 4.2.3 to generate the results in the paper. The 2.5 dimensions of the model consist of a 2D epithelial sheet with 900 non-motile epithelial cells and a 2D interstitial plane where immune cells are represented, move, and interact with the environment. The boundary conditions are periodic in the y and x directions of the simulation and use Von Neumann boundary conditions in the z direction. Diffusion-decay of cytokines and extracellular virus occurs in this interstitial plane. Together the planes represent a patch of (epithelial) lung tissue of size 0.9mmX0.9mm, represented by 90X90X2 pixels.

We made several simplifications to the biological description of the system, *e.g.*, the model does not include tissue recovery or creation and release of antibodies [1]. We started the simulation with one infected cell in the patch (out of 900 cells), and we ended our simulations at day 14. The parameter values used by the Sego-Aponte-Gianlupi model are tabulated in my Appendix A.2 Tables A.2, A.3 and A.4, and in the original paper [1]. The Sego-Aponte-Gianlupi model [1] includes only a single immune cell type that aggregates behaviors exhibited by different immune cells, mainly NK-cells and CD8+ T cells. The immune cells chemotax up the local cytokine gradient, kill infected cells on contact, and release a cytotoxic agent depending on the severity of the infection. After exposure to local cytokine, immune cells start to release cytokine, taking part in the cytokine signaling. As the modeled immune cells present behaviors of NK- and T-cells, the adaptive immune system is partially represented. In Sego-Aponte-Gianlupi's model, only one cytokine is used to represent all the cytokines involved in immune signaling.

The cytokine level and the number of immune cells in the infection zone define the system's overall pro- or anti-inflammatory state of the system. If the system is in the pro-inflammatory state, there is a stochastic chance of seeding new immune cells in the domain. If the immune system model is in the anti-inflammatory state, the simulation may remove immune cells through a stochastic process.

In our paper for the Sego-Aponte-Gianlupi model [1], we characterized possible end states for the simulated tissue domain. By varying the infectivity of the virus and the strength of the immune response, we simulate situations where the infection sweeps the tissue, cases where disease containment is achieved, and situations in which the infection is eliminated but reoccurs (recurring infection of SARS-CoV-2 has been observed *in vitro* [15]). We also characterized a particular case of note, "*failure to infect*," a situation where the infection does not spread significantly beyond the initially infected cells. This situation happens because of the stochasticity of the simulation, *e.g.* if an immune cell is seeded near the only infected cell and kills it, or if the infected cell dies before releasing infectious virus to the environment and the infection ends immediatly.

2.2.1 Epithelial cells in the Sego-Aponte-Gianlupi model

The epithelial cells in the Sego-Aponte-Gianlupi model are, as mentioned, organized in a 2D epithelial sheet of 900 cells (see Figure 2.1) and are not motile. The epithelial cells can be in one of four states: uninfected, infected (eclipse phase), virus-releasing and dead.

Each epithelial cell agent has an intracellular viral life-cycle model, and can be infected with diffusing virus (see Section 2.2.2), making them transition from the uninfected state to the infected state. Infection happens due to virus binding to epithelial cell receptors and being internalized (see Section 2.2.3), the now bound receptor is no longer available for internalization of new virus. The cells can die in several ways, the intracellular levels of viral particles may kill them, the immune cells can send an apoptosis signal to them, and the immune cells can release a cytotoxic agent. During the first infection phase (infected in eclipse) virus is assembled but not released by the cell. Then the cell transition to the viral release phase and the rate of release increases exponentially and eventually saturates.

2.2.2 Diffusion of Virus, Cytokines, and Oxidizing Chemical in the Sego-Aponte-Gianlupi Model

The Sego-Aponte-Gianlupi model has 3 different diffusive species: virus, cytokine, and a cytotoxic oxidative agent. They diffuse and decay extracellularly according to Fick's law over the simulated latice, and reach different cells. Virus can be internalized by the cells, cytokine triggers the immune response, and the oxidative agent kills epithelial cells.

The equation for viral diffusion is,

$$\frac{\partial c_{vir}(p,t)}{\partial t} = D_{vir}\nabla^2 c_{vir}(p,t) - \gamma_{vir}c_{vir}(p) + \frac{1}{\nu(\sigma(p))}\left[\rho(\sigma(p)) - \upsilon(\sigma(p))\right] \ . \tag{2.1}$$

Where $c_{vir}(p,t)$ is the concentration of virus at pixel $p = (i,\, j)$, D_{vir} is the virus diffusion constant, γ_{vir} the virus decay rate, $\sigma(p)$ the cell at pixel p, $\nu(\sigma(p))$ the volume of cell $\sigma(p)$, ρ the amount of virus being released by cell $\sigma(p)$, and υ the amount of virus being uptaken by cell $\sigma(p)$. The uptake amount (υ) is defined by Equation 2.5 and the release amount(ρ) by Equation 2.11. Uptake and release is done uniformly over the cell pixels.

Cytokine diffusion is governed by,

$$\frac{\partial c_{cyt}(p,t)}{\partial t} = D_{cyt}\nabla^2 c_{cyt}(p,t) - \gamma_{cyt}c_{cyt}(p,t) + s_{cyt}(\sigma(p),t) \ , \tag{2.2a}$$

$$s_{cyt}(\sigma(p),t) = s^{\tau}_{max}(\sigma(p))\frac{\left(c_{sig}(\sigma(p),t)\right)^{h_{cyt}}}{\left(c_{sig}(\sigma(p),t)\right)^{h_{cyt}} + \left(V^{\tau}_{cyt}(\sigma(p),t)\right)^{h_{cyt}}} - \omega^{\tau}_{cyt}(\sigma(p),t) \ . \tag{2.2b}$$

Where $c_{cyt}(p,t)$ is the concentration of cytokine at pixel p, D_{cyt} is the cytokine diffusion constant, $\gamma_{cyt}c_{cyt}$ represents the amount of cytokine leaving the simulated region, and $s_{cyt}(\sigma(p),t)$ is the amount of cytokine secreted by cell $\sigma(p)$. The net amount of cytokine secreted by cell $\sigma(p)$ depends on the cell type (τ), the maximum secretion for a cell of

that type ($s^{\tau}_{max}(\sigma(p))$), the amount of cytokine uptaken by cell σ ($\omega^{\tau}_{cyt}(\sigma(p),t)$), a quantity $c_{sig}(\sigma(p),t)$, and the cytokine dissociation coefficient for that cell type $V^{\tau}_{cyt}(\sigma(p),t)$. $c_{sig}(\sigma(p),t)$ is the amount of assembled virions for infected epithelial cells (see Equation 2.10), and the amount of cytokine the cell is exposed to for immune cells. h_{cyt} is the Hill coefficient for this equation, set to 2.

The oxidizing chemical diffuses according to,

$$\frac{\partial c_{oxi}(p,t)}{\partial t} = D_{oxi}\nabla^2 c_{oxi}(p,t) - \gamma_{oxi}c_{oxi}(p,t) + s_{oxi}(\sigma(p)) \ . \tag{2.3}$$

Where $c_{oxi}(p,t)$ is the concentration of the agent at pixel p, D_{oxi} is the agent's diffusion constant, γ_{oxi} its decay rate, and $s_{oxi}(\sigma(p))$ the secretion of the agent by cell σ occupying pixel p.

2.2.3 Viral Life-Cycle Model

The viral life cycle model begins with virus that is diffusing in the extracellular matrix entering into the cells [1]. For each cell (σ), the uptake of discrete viral particles is modeled as a stochastic process that depends on (1) the amount of virus on the cell surface ($c_{vir}(\sigma)$), (2) the cell's volume ($V(\sigma)$), (3) the number of unbound ACE2 receptors the cell has ($SR(\sigma)$), (4) the cell's initial number of unbound receptors (R_o), (5) the binding affinity (k_{on}) of the virus with the available unbound cell receptors, (6) the virus-receptor dissociation affinity (k_{off}), and (7) the time for a single uptake event to occur (α_{upt}). During a short time interval $\Delta t << \alpha_{upt}$ the probability of viral uptake (Equation 2.4a), the amount of virus internalized (if uptake occurs, Equation 2.5, υ), and the change of surface receptors (if uptake occurs, Equation 2.6) from [1] are, respectively,

$$Pr(Uptake(\sigma) > 0) = \frac{\Delta t}{\alpha_{upt}} \frac{(c_{vir}(\sigma))^{k_{upt}}}{(c_{vir}(\sigma))^{k_{upt}} + (V_{upt})^{k_{upt}}} \ , \tag{2.4a}$$

$$V_{upt} = \frac{R_o}{2k_{on}V(\sigma)SR(\sigma)} \ , \tag{2.4b}$$

$$\upsilon(\sigma) = \frac{1}{\Delta t} Pr(Uptake(\sigma) > 0) c_{vir}(\sigma) \ , \tag{2.5}$$

$$\frac{dSR(\sigma)}{dt} = -\upsilon(\sigma) \ . \tag{2.6}$$

Here, V_{upt} is the Michaelis constant of the viral concentration for which the probability of uptake is half maximum and k_{upt} is the Hill coefficient. After cellular uptake, the amount of virus that enters the cell is removed from the viral field over the cell domain.

A tri-phasic ordinary differential equation (*ODE*) system governs intra–cellular viral reproduction in each cell (σ) [1]. The first phase is the eclipse phase, during it viral release is turned off, after this phase, cells start to release virus at an increasing (but saturating) rate until the cell dies. The viral replication ODE consists of four processes and variables representing different parts of the viral replication process: unpacking virions in the cell (Equation 2.7), genome replication (Equation 2.8), protein synthesis (Equation 2.9), repackaging of new virions (Equation 2.10). Those equations, from [1], are:

$$\frac{dU}{dt}(\sigma) = \upsilon(\sigma) - r_u U(\sigma) \ , \tag{2.7}$$

$$\frac{dR}{dt}(\sigma) = r_u U(\sigma) + r_{max} R(\sigma) \frac{r_{half}}{R(\sigma) + r_{half}} - r_t R(\sigma) \ , \tag{2.8}$$

$$\frac{dP}{dt}(\sigma) = r_t R(\sigma) - r_p P(\sigma) \ , \tag{2.9}$$

$$\frac{dA}{dt}(\sigma) = r_p P(\sigma) - \rho(\sigma) \ , \tag{2.10}$$

$$\rho(\sigma) = r_s A(\sigma) \ , \tag{2.11}$$

with υ the result from Equation 2.5, r_u rate of unpacking the internalized virus, r_{max} the maximum viral replication rate, r_t the rate of translation of viral genome into RNA templates for protein synthesis, r_p protein packing rate, r_s assembled virus release rate. The values for the parameters in equations 2.4–2.11 are in Sego's Table 1 [1] and in my Appendix A.2.

In my remdesivir work [2] (Chapter 3), I extend the viral life-cycle model model to include the effects of the antiviral used (see Equation 3.2 in Section 3.2.3).

2.2.4 Immune system model

Cytokines from the simulated domain leave the tissue and reach the lymph node component of the model. The lymph node model component determines if the simulation is in a pro- or anti-inflammatory state. If the lymph node model determines that the model is in a pro-inflamatory state immune cells may be seeded in the simulated domain, if the system is in an anti-inflamatory state immune cells may be removed from the simulated domain.

The immune cell population in the simulated domain is controlled by a dimensionless signal (Equation 2.12) and a probability of seeding/removal determined from the signal (Equation 2.13). The signal is

$$\frac{dS}{dt} = \beta_{add} - \beta_{sub} N_{immune} + \frac{\alpha_{sig}}{\beta_{delay}} \gamma_{cyt} c_{cyt} - \beta_{decay} S \ . \tag{2.12}$$

Where N_{immune} is the immune cell population in the domain, the balance of β_{add} and $\beta_{sub} N_{immune}$ control the baseline population of immune cells in the simulated domain when there's no infection, $\gamma_{cyt} c_{cyt}$ is the total cytokine that left the simulated domain (see the cytokine diffusion Equation 2.2), α_{sig} determines the strength of the signal when it reaches the lymph node, β_{delay} controls the delay of the signal from the simulated domain to the

lymph node component, and β_{decay} controls how fast the signal goes back to zero. The immune seeding/removal probability is

$$\Pr(seed) = \text{erf}(\alpha_{immune} S), \text{ if } S > 0 \ , \tag{2.13a}$$

$$\Pr(remove) = \text{erf}(-\alpha_{immune} S), \text{ if } S < 0 \ . \tag{2.13b}$$

Where erf is the error function, and α_{immune} controls the sensitivity of the system to S's value.

The immune cells are motile and enter the simulated domain in a naive (inactivated) state. They sense and uptake local cytokine and have a probability of activating which depends on how much cytokine they uptook (Equation 15 in [1]), they deactivate after 10h. Once in the activated state, they chemotax on the local cytokine gradient. Immune cells recognaize infected epithelial cells on contact by antigent recognition and induce cell death on the epithelial cell. Epithelial cell neighbors of the killed epithelial cell can die through a bystander effect.

Immune cells also secrete an cytotoxic oxidizing chemical (*e.g.*, H2O2 or nitric oxide) when they are exposed to high levels of cytokine. This oxidizing chemical kills all epithelial cells when its concentration over the epithelial cell ($c_{oxi}(\sigma)$) is above a threshold (τ_{oxi}^{death}).

2.2.5 Preliminary Antiviral treatment

In the Sego-Aponte-Gianlupi [1], we did a preliminary analysis of how a treatment could be simulated. We investigated a viral replication blocker, and a viral entry blocker with the parameter variation of the viral affinity constant (k_{on}, Equation 2.4b). At the time of writing the model, chloroquine was being seriously investigated as a possible viral entry blocker.

We focused on a viral RNA replication blocker because it is the only part of the viral

replication that has an exponential ramp up of virion production per cell (see Equations 2.7-2.10). For this preliminary implementation we modeled our drug effect as binary, either off or at maximum effectiveness. This translates into a reduction of the maximum RNA replication rate (r_{max}, Equation 2.8) during the simulation. We studied several r_{max} reductions and several treatment start times. In my Remdesivir investigation [2] (see Chapter 3), I improved the preliminary drug model to include Remdesivir's pharmacokinetics, phermacodynamics, and possible variability of Remdesivir's effectiveness on different cells.

2.3 Sego-Aponte-Gianlupi Selected Results

Using the default parameters, the Sego-Aponte-Gianlupi model [1] predicts that, although the immune response slows the infection, it still sweeps the tissue and all epithelial cells die (Figure 2.2). However, depending on the parameters used, we also saw containment of the infection by the immune system [1]. At early times, the only virus available to infect additional cells is that which is released by the initially infected cell(s). This mechanism of cyclic infection can lead to quasi-synchronous bursts of cells becoming infected early in the infection.

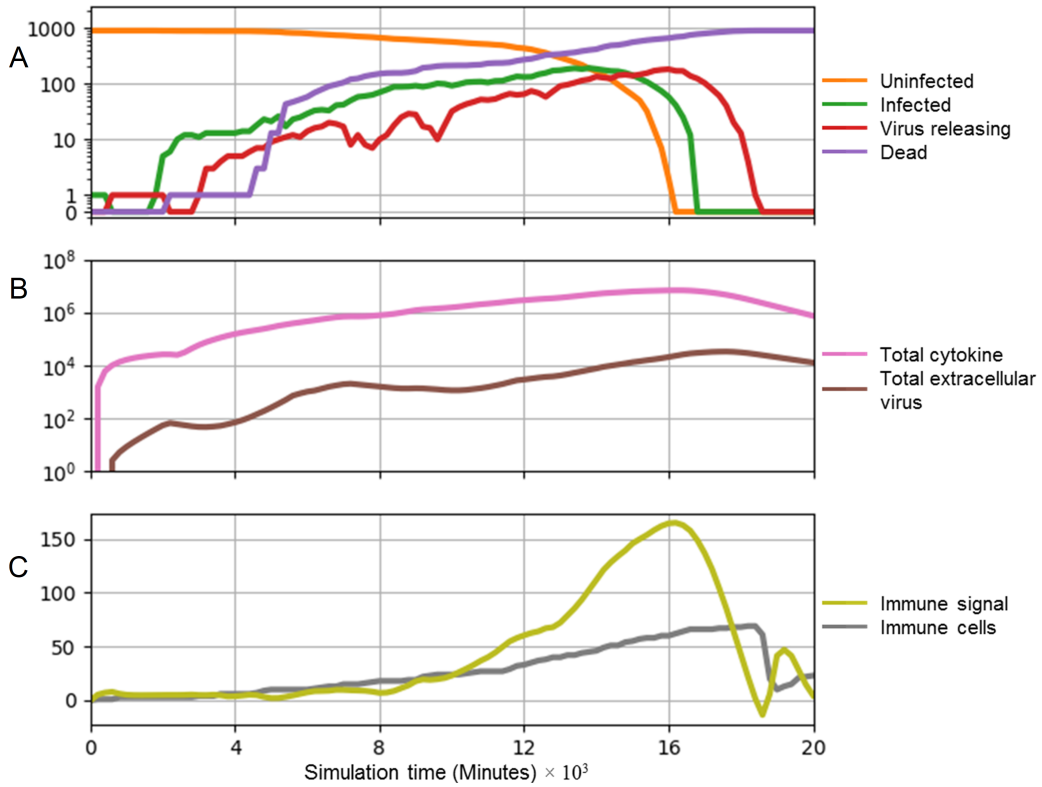
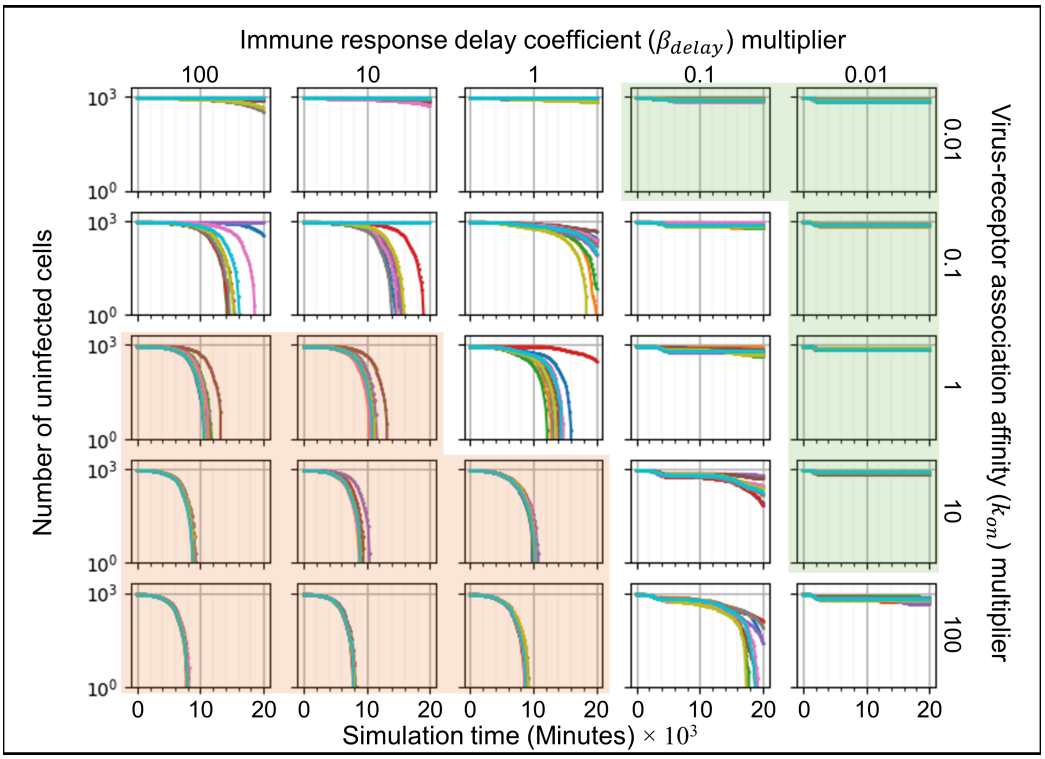
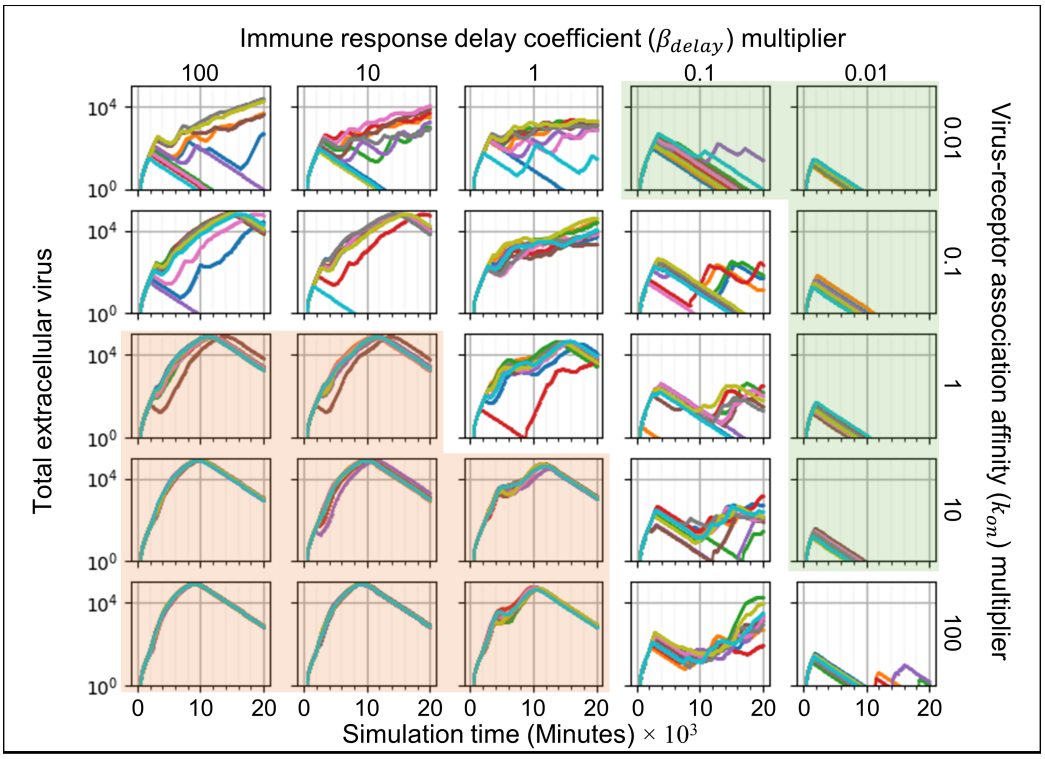


Figure 2.2: Simulation time-courses. A) Population of epithelial cells in different infection stages, uninfected in orange, infected eclipse phase in green, infected releasing virus in red, and dead cells in purple. B) Total diffusing cytokine and virus (arbitrary units), cytokine in magenta, and virus in brown. C) Immune model data, simulated domain immune cell population (grey), immune recruitment signal (S, Equation 2.12). Adapted from [1].

We performed a sensitivity analysis on the viral binding affinity constant (k_{on}, Equation 2.4b), and the immune system delay constant (β_{delay}, Equation 2.12). We found that a virus with greater affinity (large k_{on}) together with a slow immune response (large β_{delay}) result in faster infection progression and death of all simulated epithelial cells (Figure 2.3). A very infectious virus can still be contained by a fast-acting immune system.



(a)



(b)

Figure 2.3: sensitivity analysis on the viral binding affinity constant (k_{on}), and the immune system delay constant (β_{delay}). 2.3a) . Adapted from [1].

2.4 Sego-Aponte-Gianlupi Discussion

In the Sego-Aponte-Gianlupi model we were able to include several aspects of primary airway infection. We matched our simulation results to data available at the time. Our simulation predicted different outcomes for the infection and which parameters moved the simulation outcomes from one result to the other. We were also able to create a preliminary implementation of possible antiviral treatments.

The modularity of the framework was a success. We showed in the Sego-Aponte-Gianlupi paper [1] that we can easily swap the viral infection model to model hepatitis instead of SARS-CoV-2. The framework also showed its flexibility in subsequent works, such as my Remdesivir model [2], and Aponte *et al.*'s plaque growth dynamics model [16].

Although out implementation is flexible and easily modifiable, it is still limited to one modeling framework, CompuCell3D. In Chapters 5 and 4, I explain my efforts into methods that either are general to many frameworks, or to convert a model made for one framework to another.

CHAPTER 3
MULTISCALE MODEL OF ANTIVIRAL TIMING, POTENCY, AND HETEROGENEITY EFFECTS ON AN EPITHELIAL TISSUE PATCH INFECTED BY SARS-COV-2

3.1 Introduction

The COVID-19 pandemic has inspired the rapid discovery, development, and distribution of antiviral and immune-modulatory drugs and vaccines. Computer simulations of within-host response have assisted in the rapid screening of candidate drug treatments [17]. Mathematical models and their computer simulations enable us to explore alternative treatment regimens using existing drugs rapidly [18]. Models of absorption, distribution, metabolism, and elimination (*ADME*) in specific organs and the body as a whole and the pharmacokinetics of drugs within individual cells, such as at the cellular infection level, and the immune system, can be leveraged to advise clinical trials for infectious diseases [19–21].

Clinical trials of remdesivir as a possible treatment for COVID-19 followed the declaration of the 2020 pandemic by the World Health Organization [22–24]. Remdesivir is a single diastereomeric mono-phosphoramidate prodrug designed to arrest the replication of RNA viruses. Upon remdesivir administration, the patient's body generates sequential metabolic intermediates before forming the active nucleoside triphosphate, GS–443902 (GS-441524-triphosphate). The active metabolite then binds to the elongating viral RNA synthesized by RNA-dependent RNA polymerase (*RdRp*) as a nucleoside analog and blocks viral replication [25]. The first clinical trials for remdesivir were as a treatment for the Ebola virus [14, 24]. Most of these trials administered a 200 mg intravenous (*IV*) infusion loading dose followed by 100 mg IV daily infusions for five to ten days. However, the full breadth of therapeutic schedules remains unexplored, given the urgency required for drug

development during the pandemic. To that end, we modelled remdesivir and its mechanism of action (*MOA*) on a patch of lung epithelial tissue infected by SARS-CoV-2 to provide a more comprehensive understanding of the interplay of remdisivir dose and timing, and outcomes.

Even though we frame our work in the context of SARS-CoV-2 and remdesivir treatment, our methods are general to other viral infections and antiviral treatments. We have developed our own MOA model for remdesivir. There are several antiviral drugs with similar MOAs [26, 27], and previous modeling works simulated treatment with these drugs [28, 29]. Experiments on SARS-CoV-2 infection in non-human primates (rhesus macaques) and associated mathematical models have shown that an antiviral drug treatment with lower efficacy may elongate the duration of the viremic profile even if the treatment initiation is very early [30–32]. Given these results, the present model focuses on the relations of the drug potency, treatment initiation time and dose interval with the viraemia as crucial players, and performs a thorough scan of all related parameters to elucidate a reasonable treatment regimen.

Various models have described remdesivir's pharmacokinetics (PK), models ranging from one-compartment models to complex physiologically-based pharmacokinetic (*PBPK*) models [22, 23, 33]. Researches have also developed combined pharmacokinetic-pharmacodynamic (*PK-PD*) models for COVID-19. Goyal *et al.* used a two-compartment PK model for remdesivir at different potency and timing of treatment to predict how other parameters affect the disease progression and treatment efficacy [34, 35]. The authors observed that initiating antiviral treatment after symptom onset required antiviral concentrations that reduced viral production rate by more than 90% (>90% drug efficacy) to achieve a two log reduction in plasma viral load. If administration started at the time of infection (before the onset of symptoms), 60% drug efficacy achieved a similar reduction of viral load. They have also run theoretical kinetics of remdesivir drug resistance for various treatment regimens.

For the pharmacokinetics of remdesivir, we modified a PBPK model created by Gallo [33], a hybrid, full-PBPK model for remdesivir with 15 tissue compartments. Their PBPK model is very detailed and recovers remdesivir's dynamics in several tissues and plasma. Our focus is on the concentration of the active metabolite of remdesivir in lung epithelial cells; therefore, we opted to simplify Gallo's model (see Methods 3.2.2). As remdesivir is given intravenously (IV) and has a long half-life ($t_{1/2} = 30.2h$) [23], we study dosing intervals longer than one day. Treatments using more extended periods may be helpful for patients that require remdesivir administration but are not in a condition severe enough that requires hospitalization. This approach could help alleviate hospital overcrowding and could improve treatment adherence. We also aim to characterize the interplay of drug potency and schedule on infection dynamics and treatment outcomes.

The above models (coupled population, PK and PD models) can be used to study effects on infection dynamics arising from changing drug potency, half-life, and dosing schedule. Population models assume well-mixed conditions, meaning that the model exposes the entire cell population to the same amount of infectious virus at any instance. A cellular agent-based model (*ABM*) can complement such models by adding multi-cellular-scale resolution [36]. ABMs are an effective simulation technique to model a population of agents. In ABMs the agents are capable of independent decision-making according to assigned attributes and conditions [37, 38]. Cellular ABMs can introduce tissue heterogeneity to models by their very nature, as cells are individually modeled and can differ from one another, space itself is a model component [1, 4, 39–41]. A recent report on comparative biology immune ABM (*CBIABM*) has presented a model of mechanism-based differences in bat and human immune systems and discusses the consequences of these differences on disease manifestation [42]. In [4], population models of infection calibrated to experimental data were used to generate an equivalent spatially heterogeneous *ABM* of infection. The authors found that viral infectivity estimates using the ABM differed from the estimates from the population model by as much as 95% [4]. These differences in viral infectiv-

ity, or some other characteristic of the infection dynamics, could mean that a population model and an ABM calibrated to the same experimental data can significantly differ in their estimates for effective drug doses and schedules.

Furthermore, infection in a tissue starts from some discrete points of infection and spreads from them [43, 44]. Therefore, spatially heterogeneous distribution of target cell states is expected, with further disease progression near the initial infection location (necrotic sites), to regions farther from initial infection sites, where the infection has not begun. Cytokines concentrations will also be heterogeneous. Since infection can spread within a tissue even if a few cells release virus, this spatial relationship between uninfected and virus releasing cells may determine how effective an antiviral needs to be to contain the viral spread. Heterogeneity in cells reactions and drug delivery and its possible effects on disease and treatment is a topic of active study for COVID-19 [45], other diseases, and substance toxicity [46–49].

In the present ABM, we leverage our already established model of epithelial lung tissue infected by SARS-CoV-2 [1] implemented in CompuCell3D [10]. Our simulated environment models epithelial lung tissue infected by SARS-CoV-2, including cell surface-receptor (*ACE2*) affinity, intra–cellular viral replication, infectious-diffusive virus release, immune response, cytokine signaling by the epithelial and immune cells. We expand on those capabilities by incorporating a pharmacokinetic (PK) model of remdesivir and its dosing regimens, as well as a model for remdesivir's mode-of-action (*MOA*). We explore the effect of varying the time of treatment initiation (from the number of hours after the infection of ten epithelial cells), the potency of remdesivir's active metabolite (by varying IC_{50}), and the interval between doses.

As we are simulating a spatially resolved model, we can test the effects of cell-to-cell variability. The amount of drug reaching each cell in the target tissue varies. This variation can result from: (1) different availability and different distance from capillaries (microdosimetry); (2) uptake rate differences (density and dynamics of cell-surface proteins); (3)

conversion rate from prodrug to active metabolite based on intra–cellular enzyme concentrations; (4) Effect of cell-ageing on metabolic rates; (5) cell-cycle phases. To model each of these separately one needs a detailed model of cellular metabolism, life cycle and capillary structure. Therefore, in the simulations, we expose each cell to a homogeneous concentration of the antiviral drug, and combine the different possible sources of intra–cellular metabolic heterogeneity into an effective change of the uptake and elimination rates, see Methods 2.2 and 3.2.4.

Cells with internal concentrations of remdesivir-active metabolite below concentrations that control the viral replication are significant contributors to viral synthesis and release, and determine the consequent spread of infection. Their spatial distribution in the tissue is key, as those will be the regions of significant infection activity. The duration over which the concentration of the active metabolite is below the effective concentration also matters. Our previous work [1] demonstrated that if cells unblock RNA synthesis, even for a short time, the amount of functional RNA produced will be small, and one can expect reasonable inhibition of viral release.

We believe our methods can be of great use in early drug and treatment development when characterization of the drug's PK and PD are not well established. We change the remdesivir's potency and half-life in our model to investigate how those changes affect the disease progression and treatment effectiveness (see Results 3.3.3.2 and 3.3.3.3). Our heterogeneous drug metabolism model predicts that higher doses (by $\approx 50\%$ or more) are necessary to achieve the same level of treatment success compared to our homogeneous metabolism model results (see Results 3.3.3.2 and 3.3.3.4). Although the cellular level heterogeneity has not been measured experimentally, our results suggests that treatment outcomes depend on the intensity of heterogeneity (see Results 3.3.3.6). We hypothesize that the least sensitive cells to the antiviral drive the infection forwards (super-spreader cells).

This work addresses the following questions: How significant are the effects of remde-

sivir's dosing interval on treatment outcomes? What is the impact of heterogeneous cellular drug uptake and elimination on viral load (heterogeneous cellular drug metabolism)?

3.2 Materials and Methods

We sought to compare the spread of infection as represented by the total number of cells infected *versus* time and the viral load *versus* time under different remdesivir treatment schedules. Achieving this goal requires several model components: (1) viral entry into the target epithelial cells, (2) viral replication within epithelial cells, (3) release of virus and spread of infection within a tissue, (4) immune response and its effect on viral spread within the tissue, (5) the kinetics of the concentration of drug active metabolite as a function of time after dosing and (6) effect of the active metabolite on viral replication within individual infected epithelial cells.

Our computational model includes a multicellular spatial model of SARS-CoV-2 infection of a lung tissue patch composed of epithelial cells, an effective immune response module, and a minimal PK model of available active metabolite of remdesivir in individual epithelial cells. The PK model estimates changes in concentrations of remdesivir triphosphate (GS–443902) in lung epithelial cells after intravenous (IV) infusion of remdesivir in humans. We base our PK model on the Gallo model [33]; Gallo's model estimates GS–443902 kinetics in peripheral blood mononuclear cells (PBMC) following IV infusion of remdesivir. We assume that the kinetics of exposure, uptake, and metabolism of the two cell types (PBMC and lung epithelial cells) are similar, although the absolute tissue metabolite concentrations might differ. We describe each sub-model and their integration below.

3.2.1 Changes to the Sego-Aponte-Gianlupi model

We opt to simulate for 28 days even though the biological realism declines (*e.g.*, lack of tissue recovery or antibody response). Each of our ABM simulations over 28 days took 40 to 100 minutes of computation time on Indiana University's Carbonate super-computer. As

we are simulating an intervention that aims to contain the spread of infection, differentiating spontaneous results of "*failure to infect*" and effective treatment would be difficult and cumbersome. We, therefore, opted to perform more simulation replicas with the default parameters of the Sego-Aponte-Gianlupi model [1] but varying the initial number of infected cells. We ran the Sego-Aponte-Gianlupi model [1] with a single infected cell at the start out of 900 epithelial cells (same as in the original), with two infected cells out of 900, five out of 900, and ten out of 900. For each of those initial conditions, we ran 400 simulation replicas.

As mentioned, in the original model [1], the single initially infected cell (out of 900 epithelial cells) is placed in the center of the simulated domain; we randomize the position(s) of our initially infected cell(s). We can quantify the "*failure to infect*" rate and choose a starting condition that fits the goals of this paper. Results of this investigation are in Results 3.3.2, Figure 3.4. We find that starting with five initially infected cells eliminates spontaneous failures to infect without resulting in a fast sweep of the tissue by the infection (mean time to full tissue infection stays close to 15 days).

In this study, we extend the viral life-cycle model to include the effects of the antiviral modeled in 3.2.2.

3.2.2 Remdesivir physiologically based pharmacokinetic model

Gallo [33] previously published a detailed PBPK model of remdesivir. As we focus on the concentration of the active metabolite of remdesivir in lung epithelial cells, many details of the Gallo model are of limited relevance to our work. Accordingly, we build a parsimonious model to replicate the ADME and intra–cellular availability of remdesivir's active metabolite based on the time-course following IV infusion of remdesivir. We use COPASI [50] version 4.30 (Build 240) to replicate Gallo's model and build our simplified model. The original PBPK model included 6 differential equations and 12 parameters. Our model assumes that remdesivir is metabolized into its active component (GS–443902) in

the target tissue at the infusion rate, as its metabolism into the active metabolite is very rapid in the target tissue [14], then GS–443902 is eliminated by a first-order process. Figure 3.1 shows the simplified model structure. The single remaining equation for the PK model representing GS–443902 as a function of the infusion rate of remdesivir is:

$$\frac{d\left(C_{GS}\right)}{dt} = k_{app} D_{rmd} - k_{out} C_{GS} \ , \tag{3.1a}$$

$$k_{app} = \frac{k_{in}}{vol \ \ \tau_{\mathrm{I}}} \ , \tag{3.1b}$$

with k_{in} denoting the uptake rate of GS–443902, C_{GS} the concentration of GS–443902, D_{rmd} the dose of remdesivir, vol the effective compartment volume (in Liters), τ_{I} the time required to infuse remdesivir (1h), and k_{out} the excretion rate for the active metabolite. We use k_{in} as the infusion on/off switch. The values for the parameters are in Table 3.1. As part of this study, we investigate the effects of schedule on the simulated treatment, *e.g.* remdesivir is given every 48 hours instead of every 24. To keep the total amount of remdesivir administered constant, we change the infusion amount D_{rmd} with the schedule change. *E.g.* the 48h schedule uses a dose of remdesivir that is double that of the 24h schedule, $D_{rmd}(48h) = 2 \times D_{rmd}(24h)$. As shown in the Appendix Figure A.1, this simple model reproduces the C_{GS} time course within the uncertainty in the Gallo model, including underlying PBMC data as well as the European Union's compassionate use data [51].

Figure 3.1: Schematic diagram of the minimalized PBPK model of remdesivir. PBMCs are a surrogate for lung alveolar epithelial cells for GS–443902.

Table 3.1: List of parameters used for the minimal PBPK model of remdesivir.

Parameter	Value	Source
k_{in} (unit-less)	0 or 1	Fit to [23]
GS–443902 observed Half-life, $t_{1/2}$ (h)	30.2	[23]
k_{out}, GS–443902's decay rate (1/h)	$ln(2)/t_{1/2}$	
D_{rmd} (mg/day)	100 (200 for loading dose)	Doses used in clinical situations
vol (L)	38.4	Fit to [23, 51]
τ_{I} (h)	1	

3.2.3 Remdesivir mode of action (MOA) model

Remdesivir works as a nucleotide analog by binding to the elongating RNA being synthesized by RdRp [25]. To model this, we extend Sego-Aponte-Gianlupi 's [1] genome replication Equation 2.8 by modifying r_{max} as a function of the cell (σ) intra–cellular GS–443902 concentration ($C_{GS}(\sigma)$). Thus, we modify Equation 2.8 to

$$\frac{dR}{dt}(\sigma) = r_u U(\sigma) + r'_{max}(\sigma) R(\sigma) \frac{r_{half}}{R(\sigma) + r_{half}} - r_t R(\sigma) \ . \tag{3.2}$$

We also add the following equation for r'_{max},

$$r'_{max}(\sigma) = r_{max} \ \frac{IC_{50}^2}{IC_{50}^2 + C_{GS}^2(\sigma)} = r_{max} \ \frac{1}{1 + \left(\frac{C_{GS}(\sigma)}{IC_{50}}\right)^2} \ , \tag{3.3}$$

where IC_{50} refers to GS–443902's intra–cellular IC_{50}, the value for IC_{50} is in Table 3.2.

The equation for r'_{max} is a inhibitory Hill function. In order to determine the Hill coefficient, we fit a Hill equation to data of SARS-CoV-2 inhibition by remdesivir from Choy *et al.* [52] and Pizzorno *et al.* [53]. We found the value of the Hill coefficient in the range from 1 to 4 and chose 2 as the value to be used in the current study. As the intra–cellular IC_{50} for GS–443902 is unknown, especially *in vivo* [54, 55], we define a base IC_{50} of $7.897 \mu mol/L$ using our pharmacokinetics model (see sections 3.2.2 and 3.3.1, Table 3.1,

and Figure 3.3). Our investigation explores different drug potencies or doses by varying this base IC_{50} by using multipliers in the range $[0.01, 10]$ (Table 3.2).

3.2.4 Heterogeneous cellular metabolism of remdesivir modeling

The simple remdesivir PK model is present in our multicellular simulations as a component of lung epithelial cells. Each epithelial cell has a synchronized but independent copy of the model, and each cell occupies a different region of space. This method allows us to investigate the effects of heterogeneous metabolism of remdesivir by the epithelial cells. The present model avoids any complex tissue specifications by simulating a tiny patch of lung epithelial tissue. The model focuses how the intercellular heterogeneity of drug metabolism is concerned with the infection outcome. Certainly, there are many other sources of heterogeneity, which we do not consider here to minimize the cross-correlation of the stochasticity from multiple sources of heterogeneity, *e.g.*, distance from blood capillaries, tissue topology, cell age, etc.

We initialize each cell with drug metabolic parameters (k_{in} and k_{out}) at the beginning of the simulation. At the same time, to simulate metabolic heterogeneity, we modulate the metabolic parameters with random numbers selected from a Gaussian distribution. For a cell (σ_A), we draw a random number $\theta(\sigma_A)$ for each of those metabolic parameters, and we then change each metabolic parameter by multiplying it by $1 + \theta(\sigma_A)$, *i.e.,*

$$\theta_{in}(\sigma_A) = \mathcal{N}(\mu = 0, \xi = 0.25) \ , \tag{3.4}$$

$$k'_{in}(\sigma_A) = k_{in} \ (1 + \theta_{in}(\sigma_A)) \ , \tag{3.5}$$

$$\theta_{out}(\sigma_A) = \mathcal{N}(\mu = 0, \xi = 0.25) \ , \tag{3.6}$$

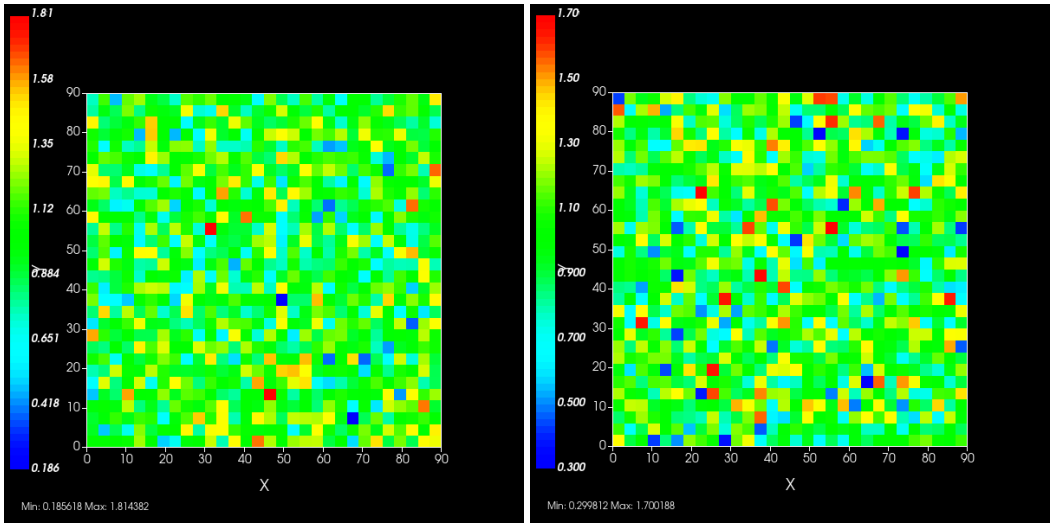
$$k'_{out}(\sigma_A) = k_{out}\ (1 + \theta_{out}(\sigma_A))\ . \tag{3.7}$$

θ is selected from the normal distribution $\mathcal{N}$ with mean μ (set to 0) and standard deviation ξ (set to 0.25). If $|\theta| > 1$, we draw a new number from $\mathcal{N}$ until $|\theta| \leq 1$. To keep the total metabolic rate over the simulated tissue patch constant, we select another cell (σ_B) to have its metabolic rates modified by $1 - \theta(\sigma_A)$ (the same θ as for cell σ_A),

$$k'_{in}(\sigma_B) = k_{in}\ (1 - \theta_{in}(\sigma_A))\ , \tag{3.8}$$

$$k'_{out}(\sigma_B) = k_{out}\ (1 - \theta_{out}(\sigma_A))\ . \tag{3.9}$$

A spatial distribution of the modified rate parameters is shown in Figure 3.2.



(a) (b)

Figure 3.2: Simulated epithelial cell layers colored by the change in k_{in} 3.2a, k_{out} 3.2b. The values displayed are relative to the base k_{in} and k_{out}, a cell colored blue in 3.2b has, *e.g.*, $k'_{out}(\sigma) = 0.3 \times k_{out}$

Typical PBPK models do not consider the effects of heterogeneous responses by the cells in the tissue, as each compartment is considered well-mixed. In reality, a cell with rapid clearance of the drug may deplete the intra–cellular antiviral and restart viral produc-

tion, release and infect neighboring cells. The difference between our model and traditional PBPK models is precisely that our model addresses this spatial heterogeneity issue.

Table 3.2: List of parameters for the ABM and PD models as well as parameters varied for the treatment effectiveness investigation

Parameter	Values used
Total epithelial population	900
Number of initially infected cells	5
Treatment initiation delay (day)	0, 1, 3
Time between antiviral doses	8, 12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144
Remdesivir doses (rescaled to match the schedules) (mg)	25, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600
Base IC_{50} ($\mu mol/L$)	7.897
Viral replication rate reduction (equation 3.3) Hill coefficient	2
IC_{50} multipliers	0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.5, 1, 5, 10

3.2.5 Simulating antiviral treatment regimens and treatment classification metrics

We simulate a 28-days daily dose regimen for remdesivir, studying the infection dynamics over that time. We initiate remdesivir intervention at different time points with a loading dose followed by a maintenance dose according to our PK model. The dose is rescaled based on the time in-between doses, *i.e.*, if the daily dose is 100mg, the dose every two days is 200mg.

We start the simulation with five infected cells, and when another five cells are infected, we consider the infection is onset. We initiate treatment 0, 0.5, 1, or 3 days after infection onset. The initial level of infection of a tissue patch can be compared with the infection of a cultured cell population. For *in vivo* infection experiments with woodchuck hepatitis virus (*WHV*), Lew *et al.* [56] found different infectivity and pathogenicity outcomes with

different inocula of similar titer concentrations. Beginning with $\approx 0.5\%$ (5 out of 900) of tissue cells infected is higher than what would be expected. However, we can imagine our infected tissue patch is surrounded by uninfected tissue that we do not simulate. For each parameter combination, we run eight simulation replicas. We classify simulation results using the median behavior of critical simulation metrics, *e.g.*, the uninfected population. Thus, we can not only differentiate effective from ineffective treatments but also create more specific classifications:

Rapid clearance (effective treatment) results from more potent and frequent treatments. Elimination of extracellular virus is achieved in less than 14 days, and there was no subsequent release of the newly generated virus (green plots in Figures 3.7, 3.8, 3.10, 3.11, and Appendix A.6).

Slow clearance (effective treatment): simulations where extracellular virus slowly decreases (over more than 14 days). In some cases new extracellular virus was produced when the antiviral concentration was low, generating an oscillation of the extracellular virus concentration around the decreasing trend. In some slow-clearance simulations, the extracellular virus cleared completely, then infection restarted once the antiviral levels dropped sufficiently (blue plots in Figures 3.7, 3.8, 3.10, 3.11, and Appendix A.6).

Partial containment (partially effective treatment): simulations with a mid-low potency result in a stable level of extracellular virus. Because the treatment is fairly effective, it keeps the viral load very low and the rate of infection of new cells is also very low. Therefore many target-cells remain uninfected throughout the duration of the simulation. The low viral load leads to a low continuum rate of infection of new target-cells roughly balancing the death of virus producing cells and the clearance of extracellular virus. In this scenario, the low dose treatments delay the infection of all target-cells and, thus, delays the onset of target-cell limited viral clearance. However, the treatment by itself is insufficient to eliminate the virus completely during the period simulated. Clearance of this low number of infected cells would likely be achieved by the adaptive immune response, which we do

not model. Note that if we ran the simulation long enough we would achieve either target cell-limited clearance or elimination of the virus by the drug. We do not imply that drug treatment makes the infection worse. The rough equilibrium of infection of new cells and viral clearance implies that these simulations are on the transition line separating effective from ineffective treatment. For a longer time between doses, we observe extracellular virus levels oscillating around a steady value (black plots in Figures 3.7, 3.8, 3.10, 3.11, and Appendix A.6).

Widespread infection (ineffective treatment): simulations resulting from sufficiently low potency or large periods in-between doses has the extracellular amount of virus increase during the treatment simulation. Treatment in these cases does not impede infection of all cells and subsequent death of the simulated epithelial tissue occurs. Some simulations have viable cells at the end of simulation with the amount of virus still increasing. Presumably, if run for a longer time, these simulations would also result in complete infection and death of the tissue. This category includes simulations with an observed decrease in extracellular virus towards the end of the simulation; this occurs after the death of all epithelial cells as virus production ceases (target-cell limited clearance) (red plots in Figures 3.7, 3.8, 3.10, 3.11, and Appendix A.6).

To classify the simulation results among these classes, we first check the median over simulation replicas of the uninfected cell population at the end of the simulation. If they number less than ten or less than half the median uninfected population at the start of treatment, we classify the treatment as ineffective with widespread infection. We then look at the median viral load; if it goes below a threshold and does not rise above it, we classify the results as "effective treatment" with rapid clearance (if cleared in less than 14 days) or slow clearance (otherwise). If the median viral load rises back to levels above the threshold, we look at the median viral load peaks trend. If the peaks are trending to higher levels, we classify the simulation results as ineffective treatment with widespread infection. If the trend is near zero (stable), we classify the results as partial containment. If we observe

decreasing trends, we classify the results as slow clearance. Greater detail regarding the definition of these metrics are in Appendix A.3.

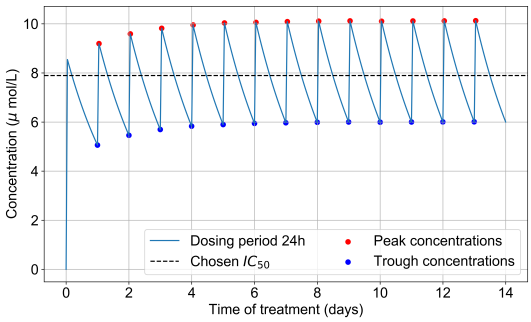
3.3 Results

3.3.1 Remdesivir PK model

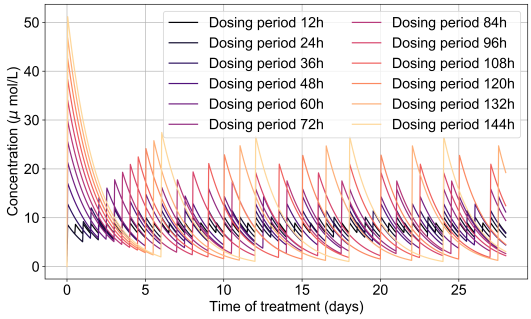
We first calculate the intra–cellular metabolite concentration over time as a function of dose and dosing schedule. The critical information is the time to accumulate to IC_{50} and how long the concentrations stay above and below values that effectively shut down viral replication.

The intra–cellular IC_{50} of GS–443902 is unknown [54, 55]. To define an IC_{50} for GS–443902 in our simulations and to characterize the GS–443902 concentration time-course and accumulation in our PK simulation we first simulated our PK model to 14 days. We then analyzed the peaks and troughs of the concentration and set the peak-troughs midpoint to be our model's IC_{50} ($7.897 \mu mol/L$, Table 3.2).

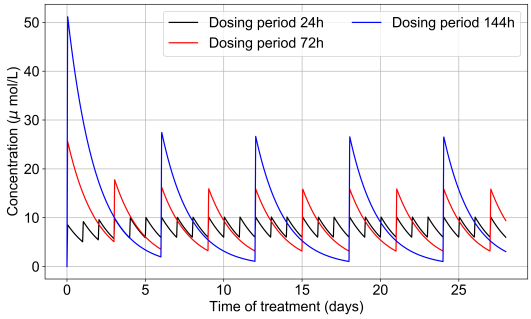
We then assessed the systems-level effects of different potencies of remdesivir. Clinically, drugs are usually given to achieve plasma concentration of the drug about 5-10 times the IC_{50} [57]. For the potency investigation, we multiply the base IC_{50} by a set of multipliers (in the range of 0.01 to 10, Table 3.2). We explore dosing regimes in which remdesivir effectively shuts off viral production in all cells; other situations where it leaves a significant number of cells releasing virus at all times; and intermediate situations. Part of our investigation is about the effect of the dosing schedule. We have included the PK concentration profiles to 28 days for all schedules investigated in Figure 3.3b.



(a)



(b)



(c)

Figure 3.3: 3.3a Concentration of GS–443902 (remdesivir's active metabolite) for a 14 days treatment with a 200 mg loading dose and 100 mg subsequent daily doses (IV infusion) is obtained by solving Equations 3.1a and 3.1a. Concentration peaks (red) and troughs (blue) are pointed out, their mid-point (dashed line) is our base IC_{50}. 3.3b Concentrations of the active metabolite, GS–443902, in PK simulations for the different dosing regimens of remdesivir, the doses are rescaled to keep the total average amount of remdesivir given over 24 hours constant. 3.3c Some selected PK profiles from 3.3b.

3.3.2 Variability of outcomes in Sego-Aponte-Gianlupi 's model

To understand the effect of treatment, first, we quantify the natural variability of infection progression outcomes in the original model (without any treatment simulation). This variability depends on model parameters, which we do not change from the original Sego-Aponte-Gianlupi model [1] and on the initial number of infected cells.

In the Sego-Aponte-Gianlupi model [1], as discussed previously in Section 2.2, simulations using the base parameters result infection sweeping in the the tissue. However, some

simulations using the default parameters show "*failure to infect*", in which the initially-infected cells die before releasing enough virus to infect a substantial number of the remaining cells. This may happen because the initially infected cells are killed by immune system cells, or by viral stress, soon after the simulation starts, while it is in the eclipse phase. The simulation, therefore, ends with almost all cells uninfected. Sego *et al.* [1] also demonstrated that their model is capable of immune containment of the infection depending on the simulation parameters.

These "*failure to infect*" results are problematic when we start our treatment investigation. We cannot distinguish for any individual simulation replica in which the infection failed to spread, whether it failed because of the treatment or whether it would have failed to spread regardless of treatment. Distinguishing these cases from effective treatment would require a vast number of replica simulations. We significantly reduce the number of replicas required per parameter set by selecting initial conditions that guarantee widespread infection in the absence of treatment (*i.e.*, eliminating "failures to infect").

As mentioned, we run Sego-Aponte-Gianlupi model [1] with a single infected cell at the start out of 900 epithelial cells, with two out of 900, five out of 900, and ten out of 900. To quantify the prevalence of "failures to infect," we run 400 simulation replicates for each of those initial conditions. With a single initially infected cell, 15.75% of the replicas resulted in failures to infect; with two initially infected cells, there was a 1.25% rate of "*failure to infect*"; and with five and ten initially infected cells, there were no failures to infect. With a single initially infected cell, the mean time for widespread infection of the simulated patch was 16.04 days (excluding failures to infect), with two initially infected cells, the mean was 27.4 days (excluding failures to infect), with five 18.6 days, and with ten the mean time to full infection was 11.4 days (see Figure 3.4).

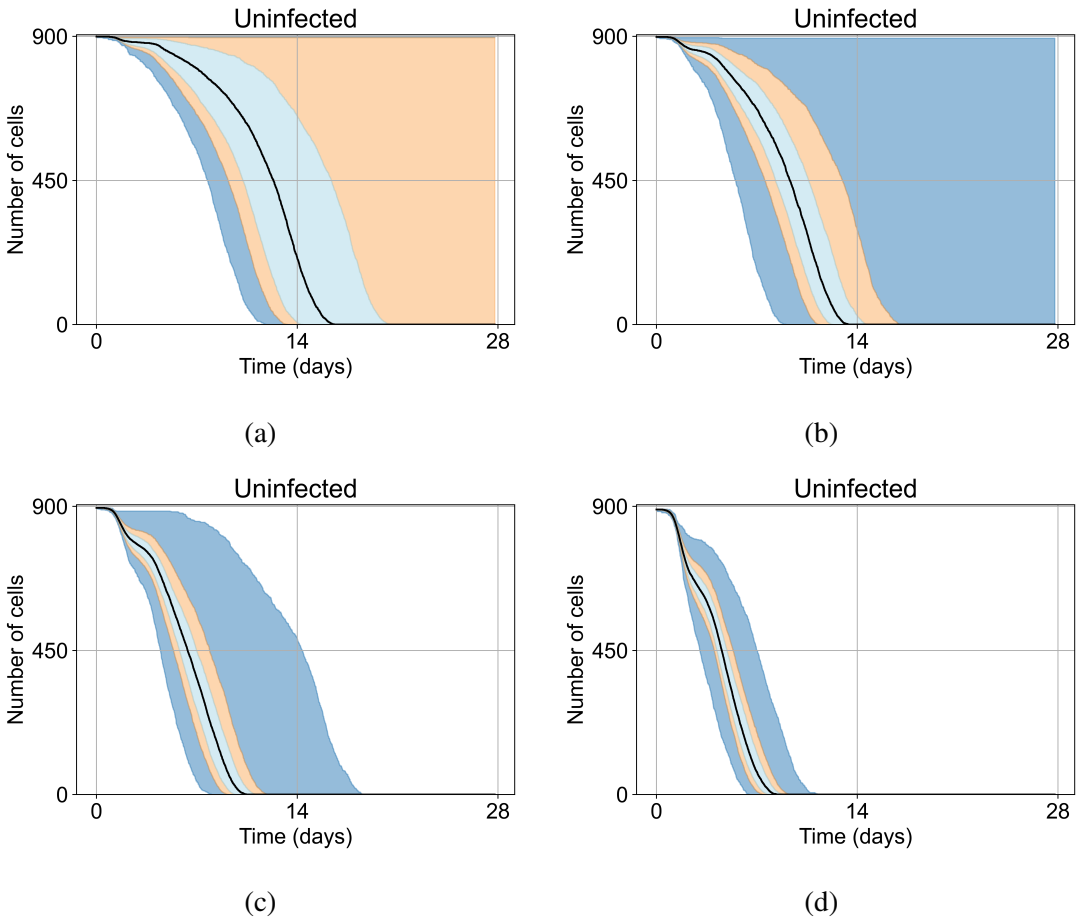


Figure 3.4: Uninfected cell populations for 400 replicas of the Sego-Aponte-Gianlupi model [1] are shown using Sego-Aponte-Gianlupi 's default parameters [1]. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. 3.4a) Simulations start with 1 initially infected cell and 63 simulations result in "*failure to infect*" (15.75% of replicas), the 90th quantile includes the upper bound of the number of cells. 3.4b) Simulations start with 2 initially infected cells where 5 simulations result in "*failure to infect*" (1.25%), the 100th quantile includes the upper bound of the number of cells. 3.4c) Simulations start with 5 initially infected cells. 3.4d) Simulations start with 10 initially infected cells.

As the infection starts synchronized, different generations of infected cells were observed at early simulation times (Figure 3.4). This is observed as a sharp drop in uninfected cells followed by a less severe infection phase followed by another drop due to the first generations of infected cells releasing virus and dying while the next generation of infected cells is still in the eclipse phase or has not been infected yet. As the simulation

progresses, the infection loses its synchronization due to the stochastic effects.

We opt to carry out the remainder of our investigations with five initially infected cells. Initiating simulations with five infected cells removes the failures to infect results (making treatment comparisons easier) while keeping the time to full infection similar to the original. In Figure 3.5, we show the viral load for simulations with five initially infected cells (*i.e.*, the initial condition for our investigation) without any treatment. The viral load peaks around day 10, which is within confidence intervals of measured viral load curves of within-host experimental data. For example, our viral replication model matches data of severe cases of COVID-19 in humans from Yanqun Wang, Lu Zhang, *et al.* [58]. In [58], patients with severe COVID-19 had a viral load peak 10-15 days after symptom onset. Those patients also had detectable virus after 20-25 days of symptom onset. In truth, viral load data for SARS-CoV-2 is still messy, especially with the several variants (of concern or not) in circulation (any of which could have different viral kinetics), as seen in Wölfel, Roman, *et al.* [59]. In any case, in this work we use SARS-CoV-2 and remdesivir as framing devices, not as the end goal of our simulations. The viral loads, viral production AUC and infected populations in untreated simulations with other initial conditions are in the Appendix A.5.

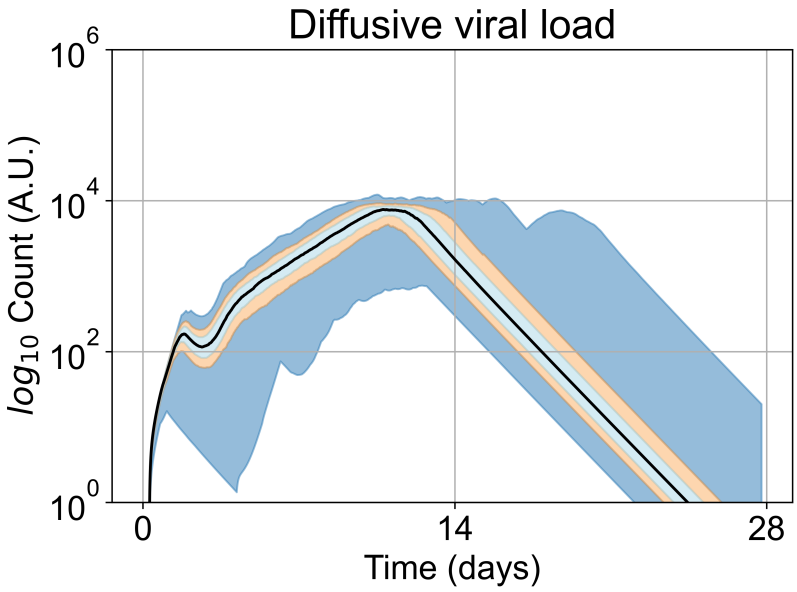


Figure 3.5: Extracellular viral load curve for untreated simulations with five initially infected cells in the tissue patch.

3.3.3 Predictive treatment outcomes

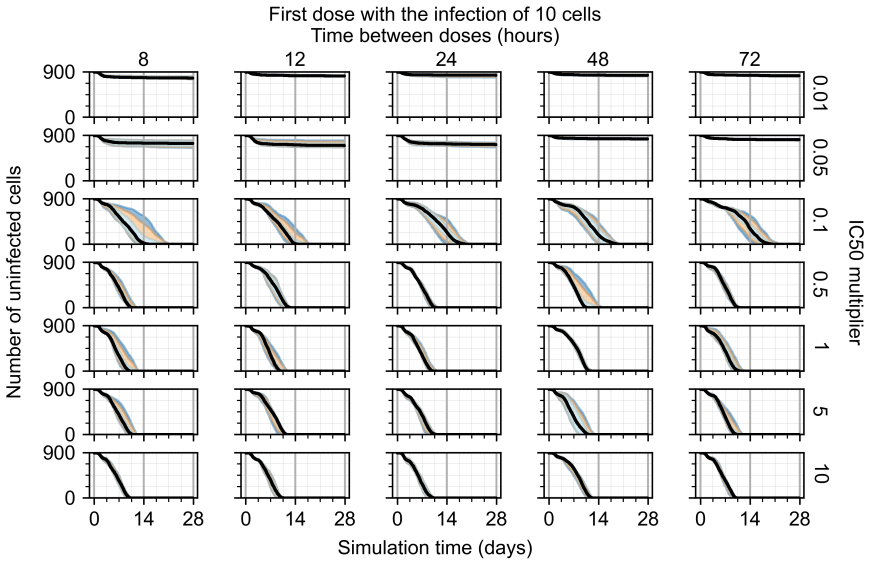
We investigate the effects of frequency, potency, active metabolite half-life, treatment start-time, and tissue heterogeneity on antiviral treatment outcomes for a viral infection on lung epithelial tissue. We model SARS-CoV-2 and remdesivir specifically, our methods are generalizable to other viruses and antivirals. We first investigate the case of homogeneous tissue, beginning with a coarse variation of the IC_{50} multiplier and dose-interval, (in Section 3.3.3.1). We then do a finer parameter investigation of those parameters to define the precise boundary of effective and ineffective treatment (see Results 3.3.3.2). We then investigate the effects of having a shorter half-life for the active component (see Results 3.3.3.3). Finally, we investigate the heterogeneous metabolism of GS–443902 by epithelial cells (see Results 3.3.3.4).

3.3.3.1 Coarse parameter variation

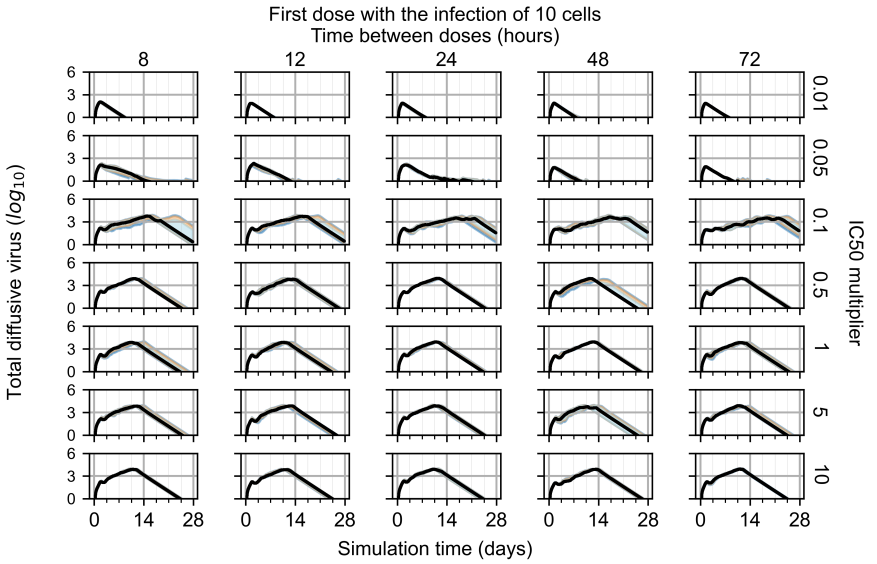
For the coarse parameter investigation, we choose 0.01, 0.05, 0.1, 0.5, 1, 5, and 10 as the IC_{50} multipliers and 8, 12, 24, 48, and 72 hours as the dosing periods (Figure 3.6). At this point of the investigation, we do not explore the effect of delaying treatment initiation, all treatments start with the infection of 10 cells, and the cellular metabolic rates are homogeneous.

The most informative metric to distinguish effective from ineffective treatments is the amount of extracellular diffusive virus. This metric allows us to define subcategories of effective / ineffective treatment. Not surprisingly, we observe that larger IC_{50} multipliers (lower potencies) and larger periods between doses result in higher levels of extracellular virus.

In this coarse investigation, we see that the simulated system transits from effective to ineffective treatment when the IC_{50} multiplier changes from 0.05 to 0.1 (Figure 3.6). However, we do not see dosing interval effects here.



(a)



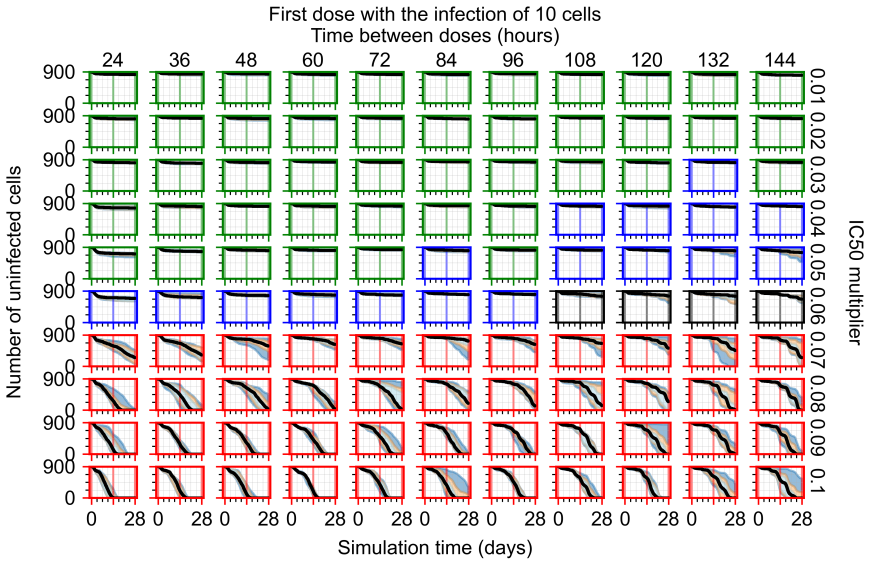
(b)

Figure 3.6: Coarse parameter investigation (10 replicas of the treatment simulation). Treatment starts with 10 infected cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Treatments with an IC_{50} multiplier < 0.055 contain the infection, while treatments with IC_{50} multiplier ≥ 0.05 do not. The top two rows show a reduction of viral load due to treatment, while in the lower two rows the decrease is due to all cells being dead 3.6a Uninfected cell population. 3.6b Extracellular diffusive virus; *y*-axis in log scale, exponent values as tick-marks.

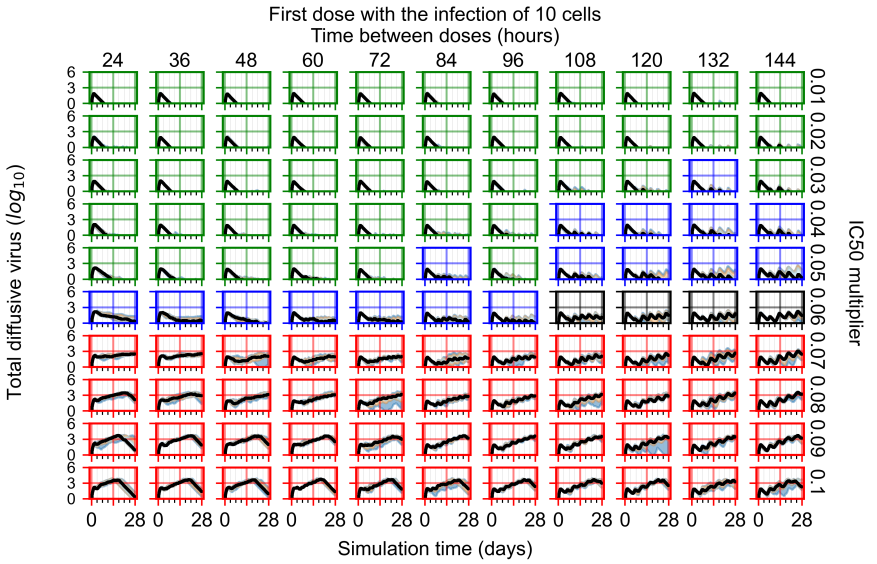
3.3.3.2 Fine parameter variation

After exploring the effects of coarse model behavior, we focus on the transition region between containment and no containment. IC_{50} multipliers of 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, and 0.1 and dosing periods of 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, and 6 days are explored (Figures 3.7 and 3.8).

The results presented here are the measurements of uninfected population and diffusing virus for simulations where treatment starts with the infection of ten epithelial cells (Figure 3.7) and simulations where we delay treatment initiation by three days (Figure 3.8). For figures of the other metrics (number of dead cells, cytokine levels, *etc.*) and simulations where treatment starts 12 hours and one day after the infection of ten epithelial cells see Appendix A.6.1. We investigate the effects of diminishing GS–443902 's half-life by 50% and by 75% in Section 3.3.3.3 and Appendices A.6.2 and A.6.3. And the effects of heterogeneous metabolism of GS–443902 in Section 3.3.3.4 and Appendix A.6.4.



(a)



(b)

Figure 3.7: Treatment starts with the infection of 10 epithelial cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.7a Uninfected cell population. 3.7b Extracellular diffusive virus; *y* axis in log scale, exponent values as tick-marks.

In Figure 3.7, there is no delay to treatment start. The system transits from effective treatment to ineffective treatment upon changing the IC_{50} multiplier from 0.05 to 0.07. We also see the effects of drug dosing frequency; for the IC_{50} multiplier of 0.05, the treatment

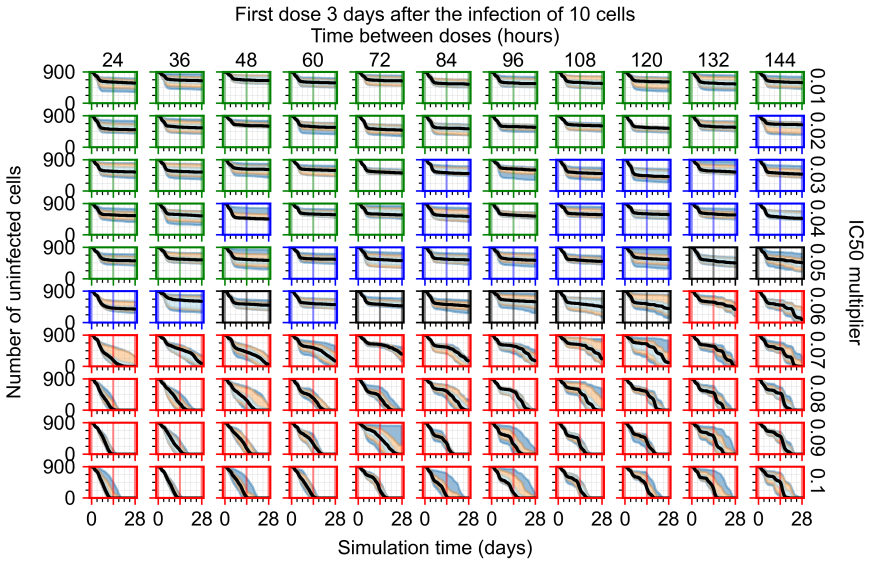
outcome shifts from fast clearance to slow clearance when we change the scheduling period from 72h to 108h. For the IC_{50} multiplier of 0.06, we see the transition from slow clearance to partial containment when we change the period from 96h to 108h.

In Figure 3.8, we delay treatment by three days. We can see that the system traverses from effective treatment to ineffective treatment upon changing the IC_{50} multiplier from 0.04 to a multiplier of 0.06. Here the effects of drug dosing frequency are more pronounced, with several transitions happening for a single IC_{50} multiplier. For instance, with an IC_{50} multiplier of 0.05, the simulated treatment moves from fast clearance to slow clearance to partial containment as time in-between doses becomes progressively longer.

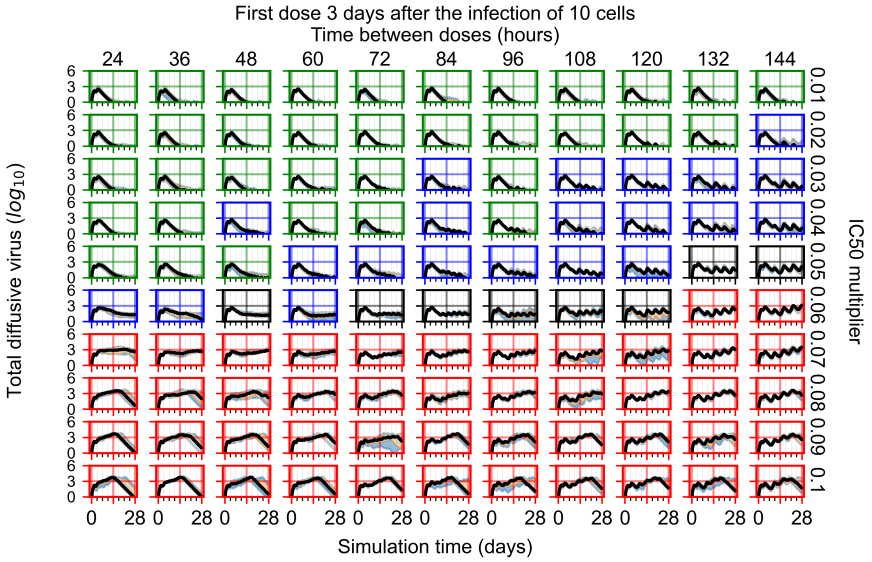
In the cases of infrequent dosing or mid-high-potency treatments, we observe a situation in which extracellular virus concentrations fall below the "cleared" threshold and then reappear. The viral resurgence happens if infected cells remain and the level of remdesivir between doses falls to a point where viral replication can restart after the clearance of the extracellular virus. Even with reappearance, however, levels of the extracellular virus do not increase beyond the first peak in concentration.

We observe an effect that seems counter-intuitive for simulations with longer inter–dose periods and intermediate potencies. There is a more significant decrease in extracellular virus for the first few days of treatment, indicating a (temporarily) more effective treatment than for the same potency with more frequent dosing. This occurs because the duration of viral replication is constrained by the duration that the antiviral concentration is above the effective concentration that inhibits viral replication. This time period is longer for larger doses. For longer inter–dose intervals the amount per dose is larger (for an inter–dose interval of n days, the dose D' is n times the amount of the standard dose (D), *i.e.*, $D' = n \times D$). Thus, the duration of inhibition of viral replication after a dose is longer for longer inter–dose intervals. The fact that the loading dose, D_L, is twice the regular dose; $D_L = 2 \times D'$, makes this effect even stronger for the initial inter–dose period. In particular, it is possible that for a longer dose interval, the drug concentration never decreases below

the effective concentration between the first and second doses, but decreases below the effective concentration during the intervals between all subsequent doses.



(a)



(b)

Figure 3.8: Treatment starts three days post the infection of 10 epithelial cells. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.8a Uninfected cell population. 3.8b Extracellular diffusive virus; *y*-axis in log scale, exponent values as tick-marks.

We show spatial configuration snapshots from replica simulations of the virtual tissue

patches for the four classes in Figure 3.9.

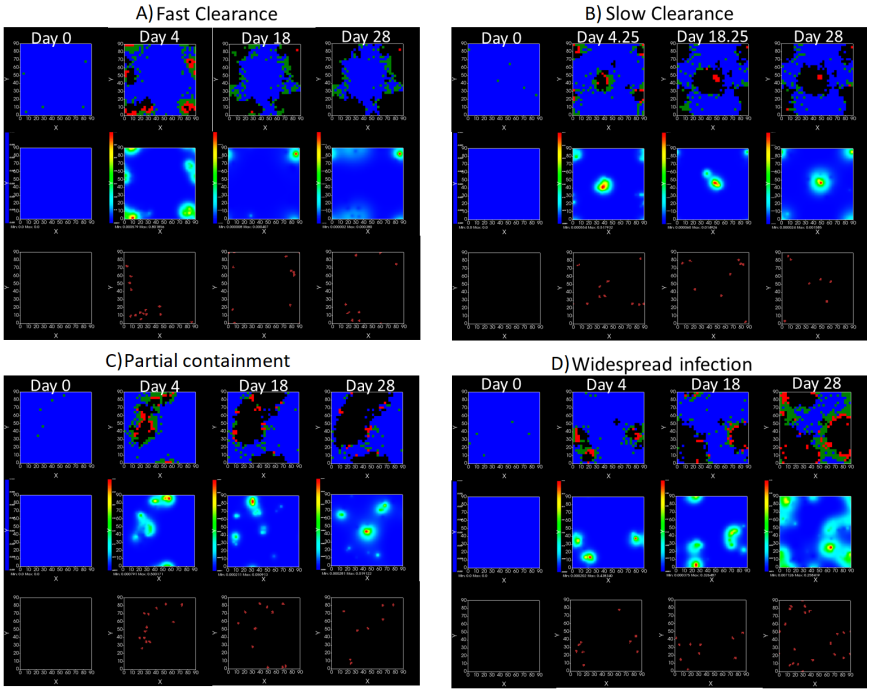


Figure 3.9: Replica snapshots of the tissue patch are shown for different treatment classifications. In all the cases the top row is the epithelial layer (blue uninfected cells, green infected cells in eclipse phase, red infected cells releasing virus, black dead cells), middle row is the extracellular virus concentrations (high concentration in red, low concentration in blue), and the third row is the immune cell layer (immune cells in red, extracellular environment in black). **A)** fast clearance (36h dosing period, 0.01 IC_{50} multiplier), **B)** slow clearance (84h dosing period, 0.05 IC_{50} multiplier), **C)** partial containment (84h dosing period, 0.06 IC_{50} multiplier), **D)** widespread infection (108h dosing period, 0.07 IC_{50} multiplier). In all the cases snapshots are shown at the start of the simulation (day 0), at the start of treatment (3 days post infection of 10 cells), after 14 days of treatment, and at the end of the simulation (Day 28).

3.3.3.3 Faster clearing drug necessitates more potent antiviral in order to contain the infection

We evaluated the effects of increased drug clearance on tissue outcomes. Specifically, we reduced the half-life of GS–443902 by 50% (from 30.4h to 15.2h) (see Figure 3.10 and Appendix A.6.2), or by 75% (to 7.6h) (see Appendix A.6.3). With the faster clearing drug, we observe a general effect of shifting the region of effective treatment to greater potencies (smaller IC_{50} multiplier); however, there is no similar shift towards more frequent doses

schedules.

With the half-life of GS–443902 reduced to 15.2h and no delay to treatment initiation, we classify many more treatments as ineffective than treatments using the regular half-life and delaying treatment by three days.

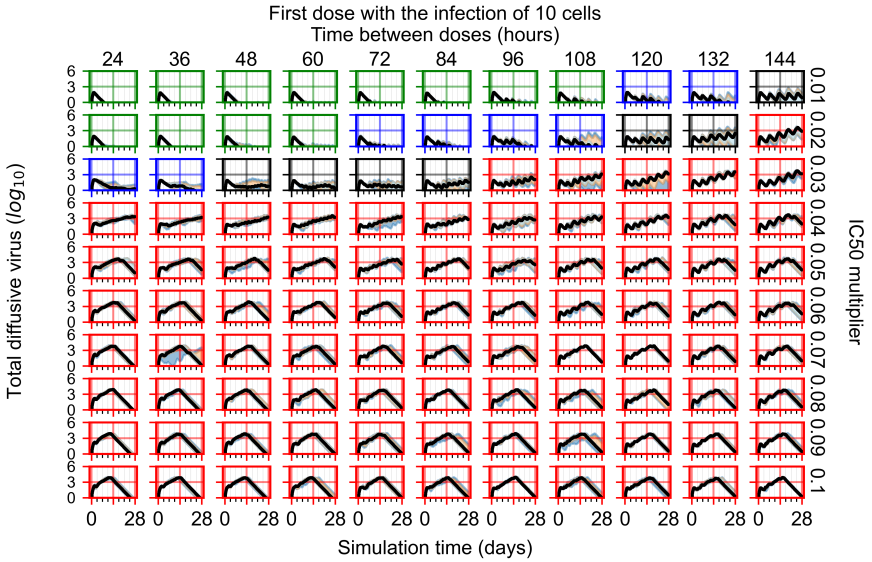


Figure 3.10: Extracellular diffusive virus populations for 8 replicas of the treatment simulation. Treatment starts with the infection of 10 cells, the half-life of GS–443902 was halved (to 15.2h). For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red.

Initiating treatment later, at one or three days post-infection of ten epithelial cells, pushed outcomes to widespread infection (see Appendix A.6.2). However, as most treatments are already ineffective with the faster clearing antiviral, few options were pushed to ineffectiveness.

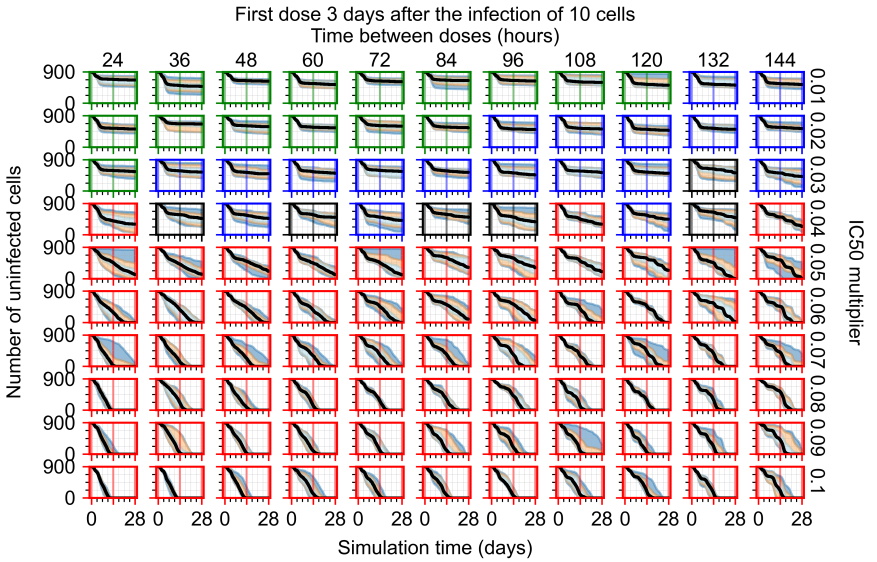
3.3.3.4 Heterogeneous cellular metabolism of remdesivir results

Now we vary the metabolic rates of the antiviral in each epithelial cell individually, as detailed in the Methods Section 3.2.4. We see that heterogeneous drug metabolism and clearance result in an overall worse treatment outcome. We believe this occurs due to

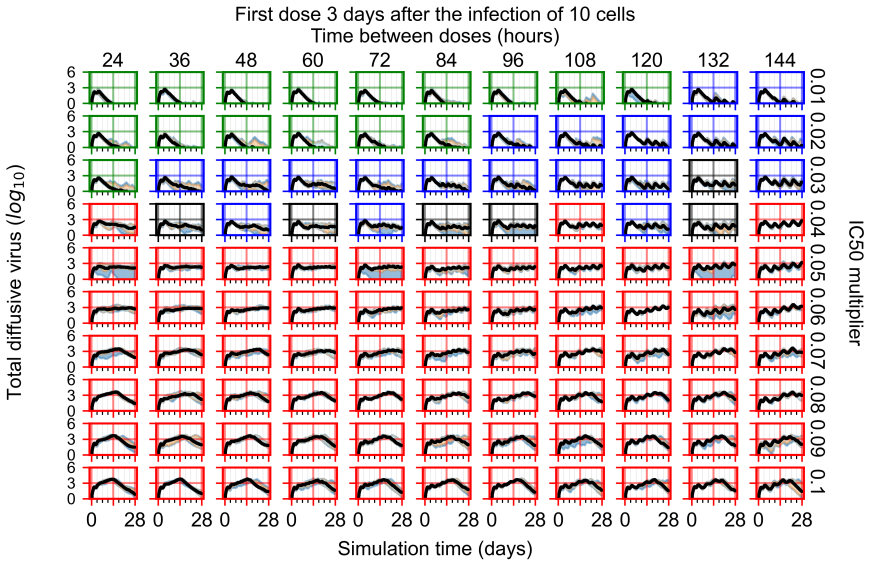
infection being driven forward by the cells that generate the most extracellular virus; we test this hypothesis in Results 3.3.3.5. If there is a region of cells with good drug metabolism (the antiviral is effective) but one among them is insensitive, the infection will progress.

For the heterogeneous cellular metabolism simulations, treatment shifts from effective to ineffective when we change the IC_{50} multiplier from 0.03 to 0.05 (Figure 3.11) instead of 0.04 to 0.06 for the homogeneous case (Figure 3.8), both when there is no delay to treatment initiation and with delay. However, when there is no delay, the 0.05 IC_{50} multiplier results in mostly partial containment, and the 0.03 IC_{50} multiplier guarantees clearance. When we delay treatment by three days, 0.05 IC_{50} multiplier results in mostly widespread infections, and the 0.03 IC_{50} multiplier is close to the slow clearance – partial containment transition. Even the best-case scenario, starting treatment with the infection of ten epithelial cells with heterogeneous metabolism, yields worse outcomes than the worst-case scenario, starting treatment three days after the infection of 10 epithelial cells, for the homogeneous case.

For the heterogeneous cell response simulations, changing treatment initiation time has a more pronounced effect than the homogeneous case, see Figure 3.11 and Appendix A.6.4.3. For instance, when we delay treatment by 1 or 3 days (Figure 3.11b and Appendix A.6.4.3), most treatments for an IC_{50} multiplier of 0.3 become almost ineffective (slow clearance) from most being classified as fast clearance when there is no delay (Figure 3.11a). For an IC_{50} multiplier of 0.05, we classify all except one treatment option as partial containment with no delay (Figure 3.11a). In contrast, we classify most infection dynamics as widespread infection with a delay of 1 or 3 days (Appendix A.6.4.3, Figure 3.11b).



(a)



(b)

Figure 3.11: Extracellular diffusive virus populations for eight replicas of the treatment simulation. Epithelial cells' metabolism and clearance varies from cell to cell. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.11a Treatment starts with the infection of 10 cells. 3.11b Treatment starts three days post the infection of 10 cells.

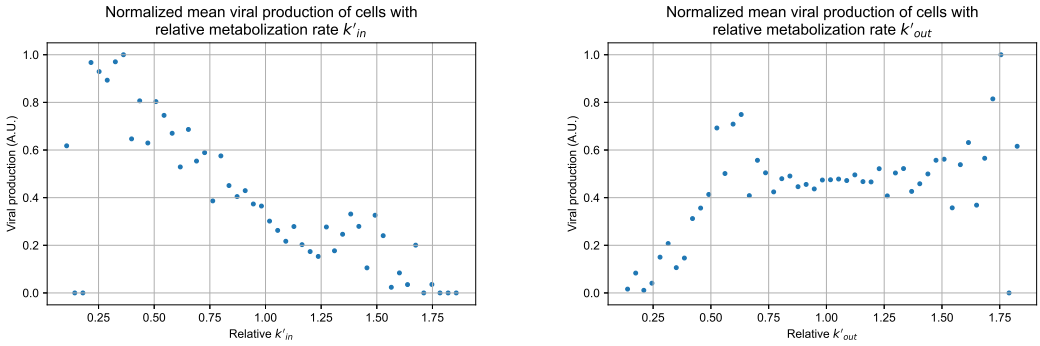
3.3.3.5 Factors responsible for negative treatment outcomes in the heterogeneous metabolism model

As cellular heterogeneous metabolic response worsens treatment outcomes, we perform simulations tracking viral production (viral AUC) by individual cells. We hypothesized that the infection is driven forwards by the cells that are least sensitive to the antiviral. We selected four parameter combinations used in Figure 3.11b (*i.e.*, simulations with treatment delayed by three days), one from each classification, to perform simulations investigating the viral production by individual cells. The parameter sets chosen are:

- Rapid clearance: 24 hours dose interval, 0.01 IC_{50} multiplier;
- Slow clearance: 120 hours dose interval, 0.03 IC_{50} multiplier;
- Partial containment: 96 hours dose interval, 0.06 IC_{50} multiplier;
- Widespread infection: 24 hours dose interval, 0.1 IC_{50} multiplier.

In these simulations, we vary either k_{in} (Figure 3.12a) or k_{out} (Figure 3.12b). We measure the per-cell viral production and see the production–metabolic rate relationship. If our hypothesis is correct, we would expect with k_{in} increase, the average intra–cellular concentration of antiviral increases subsequently, and the viral production would decrease. For k_{out}, we would expect the opposite correlation. With increasing k_{out}, the intra–cellular viral concentration decreases and viral production will increase.

As before, we run four simulation replicas for each parameter set. We then combine the replicas' per cell viral production data, separate the cells into 50 bins of the metabolic rate range (either k_{in} or k_{out}), and calculate the mean production in each bin.



(a) (b)

Figure 3.12: Mean viral production of cells versus their relative metabolic rates normalized by the maximum mean production with partial containment parameters. 3.12a Results for simulations varying only k_{in}, uses the partial containment parameter set. 3.12b Results for simulations varying only k_{out}, uses the widespread infection parameter set.

Figure 3.12 shows the results for the parameter sets that had the most evident correlation. Namely the partial containment set for k_{in} and widespread infection set for k_{out}. We see that the hypothesis of super spreader cells is reasonable, as the correlation of metabolic rates and viral production agrees as we would hypothesized. As k_{in} increases, viral production decreases and as k_{out} increases, viral production increases. Outliers are escaping the trend at the limits of the metabolic rates. Several factors explain these outliers, such as the time a cell releases virus before dying. On top of that, the distribution of rates is not uniform but a normal distribution (see Methods 3.2.4). As the number of cells with either high or low metabolic rates will be low, the *mean production* of cells with high or low metabolic rates is more likely to have fluctuations.

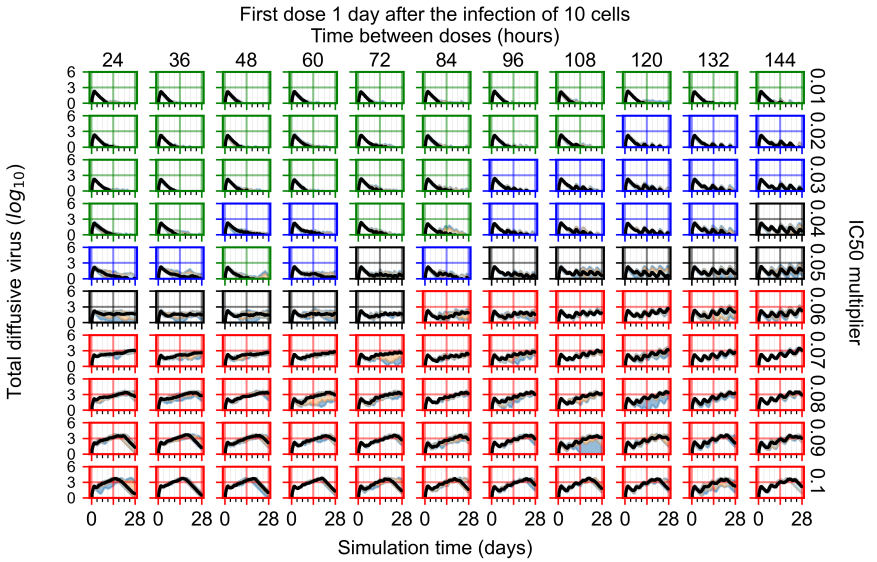
The least sensitive cells to the antiviral drive forward the infection and complement a recent result from Reinharz *et al.* [60]. They have shown that cellular heterogeneity in interferon signaling response has the effect of improving tissue outcomes. Antiviral resistance due to interferon depends on the cells that produce the most interferon (effectively warning other cells of the infection).

For the other parameter sets see Appendix A.7. As there are several sources of heterogeneity in the model, the antiviral-metabolic rate *vs.* viral production correlation can be

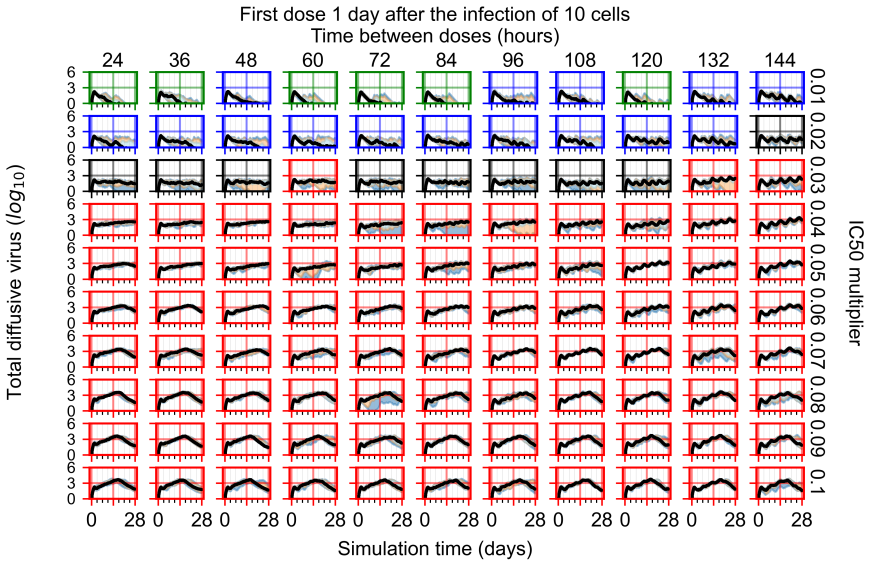
less apparent, which is the case for the results in Appendix A.7. For instance, the correlation is weak if the infection spreads fast (killing virus producing cells early) or is contained quickly (making the number of infected cells low). Weak correlation can also arise from the infected cells dying before or shortly after releasing the virus to the environment.

3.3.3.6 Effects of variability in cellular drug metabolism on treatment outcomes

To quantify the effects of variability in cellular drug metabolism on treatment outcome, we simulated tissues with different levels of variability. This was done by changing the modulation parameter of the metabolic rates, k_{in} and k_{out} in Equations 3.4 and 3.6. For the results so far, in Equations 3.4 and 3.6 we modulated the metabolic rates by drawing a random number from a Gaussian distribution with mean (μ) set to 0 and standard deviation (ξ) set to 0.25. Here we perform simulations with $\xi = 0.1$ (see Figure 3.13a) and $\xi = 0.5$ (see Figure 3.13b). With a $\xi = 0.1$ most treatment outcomes that are classified as rapid or slow clearances in the homogeneous case remain effective. In contrast, with a $\xi = 0.5$, almost all treatments are ineffective. We map the boundary of effective-ineffective treatments (Figure 3.14) by plotting the centers of simulations classified as slow clearance. We observe that the boundary moves towards lower IC_{50} multipliers (*i.e.*, more potent drugs would be needed). We also saw that the increase of metabolic variability made the boundary less dependent on the dosing interval. Other metrics for these simulations can be found in Appendix A.6.5. In summary, with a lower variability of cell metabolism, treatment outcomes were closer to those of uniform cell metabolism; whereas, with a higher variability most treatments became ineffective.



(a)



(b)

Figure 3.13: Extracellular diffusive virus populations for eight replicas of the treatment simulation. Epithelial cells' metabolism and clearance varies from cell to cell. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. Rapid clearance plot axis in green, slow clearance plot axis in blue, partial containment in black, widespread infection in red. 3.13a Cells' metabolism standard deviation set to 0.1. 3.13b Cells' metabolism standard deviation set to 0.5.

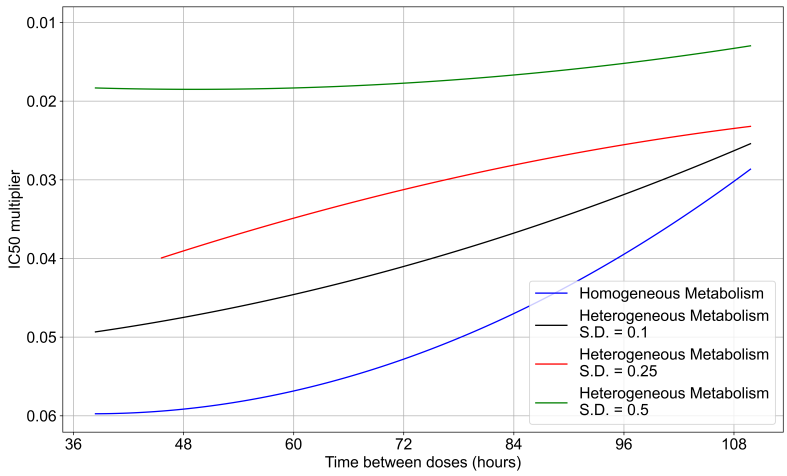


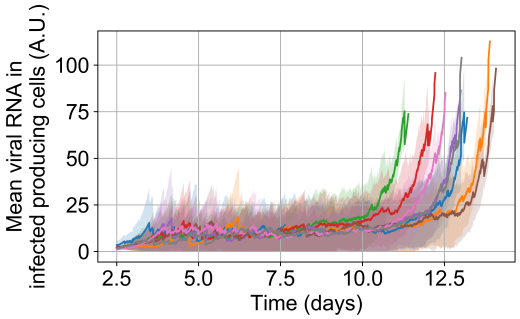
Figure 3.14: Ineffective-effective treatment transition border for different levels of metabolic variability. In blue is the homogeneous metabolism case, in black the heterogeneous metabolism case applying a modulation by a Gaussian random number with standard deviation (S.D.) set to 0.1 for k_{in} and k_{out}, in red the heterogeneous case with the Gaussian random number S.D. set to 0.25, in green with the S.D. set to 0.5.

To appreciate the effects of cellular metabolic heterogeneity on treatment outcome, we compare intra–cellular viral RNA levels in untreated cases to the same with treatment at different levels of heterogeneity (*i.e.* different ξ). Quantifying intra–cellular RNA is not straightforward, as the time of infection of different cells varies, and the total number of infected cells at a given time also varies. Therefore, we consider how to aggregate cells and measure intra–cellular RNA. We opt to measure the viral RNA levels only for infected cells that release virus to the extra cellular environment (*i.e.*, that have exited the eclipse phase of infection), as they are further along in the disease progression and will have the highest intra–cellular RNA levels. It is also important to note that growth of RNA level in cells under ineffective or no–treatment is exponential, therefore mean RNA levels can be dominated by a small number of cells until they perish due to viral production stress. We see such cases as spikes in RNA levels followed by a rapid decay in Figure 3.15.

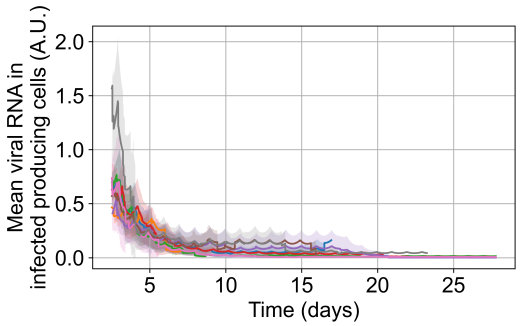
For the intra–cellular RNA level comparison we have one set of simulations with no treatment at all, and for the treated simulations we choose a set of treatment parameters that yielded results classified as rapid clearance when there is no metabolic heterogeneity,

as slow clearance with the metabolic heterogeneity ξ set to 0.1, as partial containment with $\xi = 0.25$, and as widespread infection with $\xi = 0.5$. Chosen treatment parameters are: delay treatment for one day, dose every 24h, IC_{50} multiplier of 0.05.

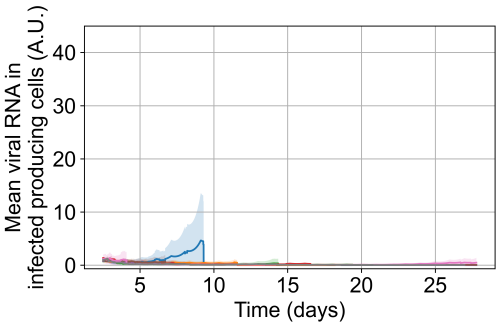
In Figure 3.15, we show the mean and standard deviation of RNA levels in virus-producing cells for 5 sets of simulations with the simulation replicas of each set plotted individually, both for the untreated case (see Figure 3.15a) and with treatment (see Figures 3.15b, 3.15c, 3.15d, and 3.15e). We want to point out that the scales of Figure 3.15's subfigures are not the same, so that the change in RNA level can be seen for all cases.



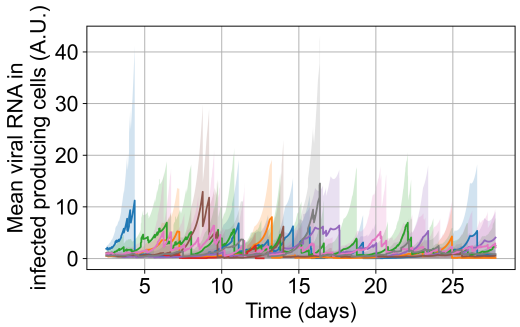
(a)



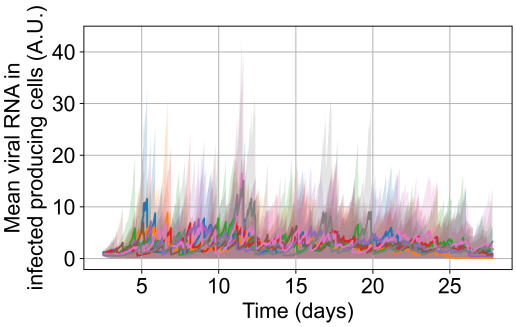
(b)



(c)



(d)



(e)

Figure 3.15: Mean RNA levels in virus-releasing infected cells (solid lines) with standard deviation (shaded regions) versus time for individual simulation replicates under different simulation options. Sub-figure 3.15a is untreated. Sub-figure 3.15b is treated with no metabolic heterogeneity, 3.15c is treated with a metabolism standard deviation of 0.1, 3.15d is treated with a metabolism standard deviation of 0.25, and 3.15e is treated with a metabolism standard deviation of 0.5. Please note that the y-range in 3.15a and 3.15b differ from the others.

Without any treatment, mean RNA levels in virus-producing cells stay stable at a level of 12 arbitrary units until the end of the simulation, when RNA levels in the few surviving epithelial cells shoot up. Before that time there are cells with high levels of internal viral RNA but the mean level is brought down by cells that are earlier in the infection stage. With treatment and no inter–cellular metabolic heterogeneity the intra–cellular RNA levels in virus-producing cells stay low, below 2 A.U., for the entire treatment duration and RNA levels among cells are similar (low standard deviation). We also see that as treatment progresses the mean RNA levels steadily decrease, and we see a periodic effect from the dosing interval. This is to be expected when treatment is effective and all cells respond to treatment identically (see Figure 3.15b). As we increase metabolic heterogeneity (ξ) the mean intra–cellular RNA levels also increase as does the variation of intra–cellular RNA levels among cells. We also start to see spikes in the mean intra–cellular RNA levels related to cells that do not control viral replication internally (one spike for $\xi = 0.1$, several for $\xi > 0.1$), where in those cells the RNA growth is exponential. As soon as those cells die the mean intra–cellular RNA drops back down to a steady level. When we use $\xi = 0.5$ in Equations 3.4 and 3.6 the RNA levels oscillate in a similar range to that of the untreated case. In Appendix A.6.5.3 we show how changing the metabolic heterogeneity affects the average and distribution of intra–cellular drug levels in these simulations. In short, when $\xi = 0.1$ intra–cellular drug levels stay close to the homogeneous metabolism case. With $\xi = 0.25$ intra–cellular drug concentrations are more spread–out, and with some cells at levels that are double or half the mean. With $\xi = 0.5$ there are cells with 0 intra–cellular drug concentration and cells with intra–cellular concentrations as much as 3 times greater than the mean.

3.4 Discussion

Our model of antiviral treatment of COVID-19 integrates a spatio-temporal model of an infected lung tissue patch with remdesivir PK and PD models. This approach allows us to

probe the distribution, dynamics, and effects of remdesivir within a hypothetical pathological tissue structure.

This work is part of a collaborative effort. The modeling framework is extendable and modular. Models of other viruses and therapies can replace our viral model and PK-PD models.

Computational methods are a necessary complement to experimental efforts moving forward in the fight against COVID-19, future pandemics and current diseases. The combined complexities of the pathogen, disease pathology, immune response (both innate and adaptive), antiviral dynamics, and host variation, including variation in the basics units of the body, make it virtually impossible to disentangle the numerous driving forces behind infection outcomes using only animal and human data. Furthermore, experimental data are often sparse, and computational models can expand the explanatory power of limited experimental data. Our modular approach captures and integrates these dynamics to help translate biomedical mechanisms to clinical relevance.

We characterize the infection and treatment dynamics of an epithelial patch infected by SARS-CoV-2 and treated with remdesivir, at dose regimens encompassing those approved clinically. To create our model, we used reported human pharmacokinetics of remdesivir and its active metabolite and physiologically relevant within-host viral dynamics. We have considered the PK model profile of remdesivir to calculate the concentration of the drug in each cell, while assuming, at first, that the availability of drug to each cell is equal (well–mixed conditions). The lung tissue concentration of all of the remdesivir metabolites are not evident in any reported biomedical study, therefore we estimated them from their plasma concentrations using pharmacokinetic models [23, 51, 61].

We simulate multiple regimens for antiviral therapy on top of our extendable framework. We aimed to explore if the unconvincing results of antiviral trials and their clinical use could be explained by exploring the effects of changes in drug potency and schedule and some unknown possibilities. We found that the same dosing regimen and the same

parameter set different replicas can have different outcomes. Furthermore, we found that if the cellular metabolism of the antiviral changes from cell-to-cell, simulation outcomes are more dispersed, and the effective antiviral dose is greater. Altogether that means that two identical patients receiving the same identical treatment could have different outcomes. We believe this explains some of the ambiguity of the clinical trials.

The spatial model enables exploration of the effects of inter–cellular heterogeneous response to the antiviral, even with a lack of experimental data on how different cells of the same tissue may react and metabolize drug compounds differently. Therefore, we perform simulated experiments to investigate how that variation in metabolism may affect treatment outcomes. Our model predicts that cells that are less sensitive to the drug drive the infection, necessitating higher antiviral doses or the need for a more potent antiviral activity (by $> 50\%$ depending on level of heterogeneity and treatment delay) (see Results 3.3.3.2, 3.3.3.4, and 3.3.3.6). Therefore, experimental studies that investigate what are the typical metabolic variations in a tissue would enable the development of models that fit even better with reality. A pure population model of infection would not predict that cellular heterogeneity increases the antiviral potency needed for effective control of the infection, as they cannot model inter–cellular heterogeneity in space.

As expected, the model predicted that remdesivir exposure, both to adequate doses of antiviral and in the amount of time with adequate levels, is a crucial determinant of treatment outcome, implying that increased dose amount would improve treatment outcome. We show that suboptimal exposure to the simulated antiviral inside simulated infected epithelial tissues leads to regrowth of viral load between doses and may contribute to persistent COVID-19. This result aligns with the reported experiments in non-human primate, rhesus macaques [30–32].

However, we do not model drug-induced toxicity, which is a concern at high doses and limits the clinically safe dose. We also predicted the continued significance of host mechanisms during treatment, such as metabolic clearance of the antiviral (see Results 3.3.3.5).

Identification and ranking of these host mechanisms identifies potential targets for therapy development [62, 63]. Potential strategies include treatment utilizing a neutralizing antibody cocktail or convalescent plasma to boost host immunity or fast viral clearance. Here, we want to comment that the present study is a mathematical and mechanistic exploration of possible scenarios. The present model simulates a tiny patch of epithelial tissue and cannot predict clinical outcomes at the whole-body level. The present model lacks a well-defined immune response that allows natural clearance of the virus. The antibody-mediated viral neutralization and antibody-dependent cellular cytotoxicity is not considered in the present model, hence the second, rapid clearance phase is not observed in the viraemic loads of our simulations. We also recognize that the present set of simulations aggregate rather than explore, the different sources of heterogeneity and variability. Future work should explore the sources of heterogeneity, incorporate the antibody therapies with an improved, well-defined immune response model, as well as of tissue recovery. In turn, improved immune response modeling will help us determine the cross-talk between antibody cocktails with the host immune response while reducing the viral burden. Because of these results, we believe that researchers must consider host mechanisms, viral load, and drug permeability as part of the design space combining immuno-modulation and antiviral treatment of COVID-19 and other viral diseases.

The main finding of this paper is the effects of heterogeneity in cellular metabolism (uptake and clearance) on viral replication. Even in the presence of spatial stochasticity, identical cellular behavior predicted better outcomes for the antiviral treatment. In contrast, including cellular heterogeneity worsened the treatment outcomes and produced results that are more similar to previous clinical trial outcomes than the homogeneous setup [64]. This discrepancy of model prediction and clinical trials may be a limitation of traditional pharmacometrics models that utilize the well-stirred assumption. As employed in this study, incorporating tissue heterogeneity may be important to improve the clinical trial simulation.

Clinical trials of antivirals for COVID-19 remain fraught with limitations, including the inability to test drugs singularly or in combinations, high cost, and the long duration of clinical trials. The most significant limitation of antiviral trials as treatment is that antivirals need to be given early in infection, which is a challenging issue for SARS-CoV-2, where there is an appreciable delay between infection and symptom onset. Animal models play an essential role in identifying new and effective regimens, but these studies are also time-consuming and costly, and they require models with human-like pathology. Here we provide a complementary systems pharmacology tool for predicting the efficacy of new drugs and regimens, allowing a rapid assessment of drug efficacy at the site of viral infection while considering cellular heterogeneity.

CHAPTER 4
TRANSLATING MODEL SPECIFICATIONS

4.1 Introduction

We build biological models to describe and investigate biological systems. Our models need to follow the scientific method, where we start from an observation about a natural phenomenon or problem, formulate a clear and specific question, create a tentative and testable hypothesis, make a prediction, and perform experiments (in our case, our experiment is *in silico*, *i.e.*, the model itself) to test the hypothesis. After doing the experiments we can analyse the results, compare them with wet-lab experiments (if they exist), come to a conclusion, suggest new wet-lab experiments, and publish our model and results.

After publication, it would be ideal if the model is reproduced using another method, after all, the modeler's choice of which platform and method to use in their investigation, if we assume the model is build correctly, should not change the results from the model. In fact, one way to check the validity of a model is to re-implement it on a platform that utilizes a different method to represent biology. Replication and reproduction are one of the most important parts of the scientific method, they guaranty the falsifiability of the experiment and hypothesis.

Do multicellular tissue agent-based models (ABMs) follow the scientific method fully? Science and computational models should be: **f**indable, **a**ccessible, **i**nteroperable, and **r**eusable (**fair**) [65]. Tissue ABMs usually follow the first two criteria, but do not follow the later two. *E.g.*, a PhysiCell [11] model cannot be reused in CompuCell3D [10]. This is a result of the fact that, currently, there's no easy and unified method to use a biological agent-based models (ABMs) developed for one platform in another.

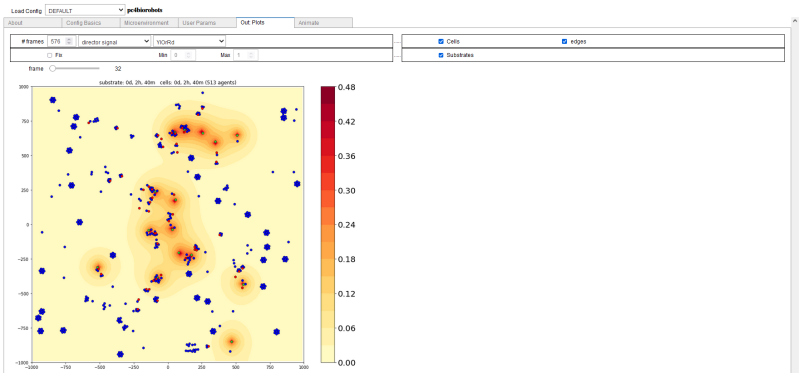
The lack of interoperability and reusability slows down research and inhibits collab-

oration, and causes a crisis of reproducibility. Theses issues could be mitigated by re-implementing a model by hand. However, re-implementing a model is a time intensive task that requires deep knowledge of the model and both platforms, therefore, it is rarely done. How can we change that?

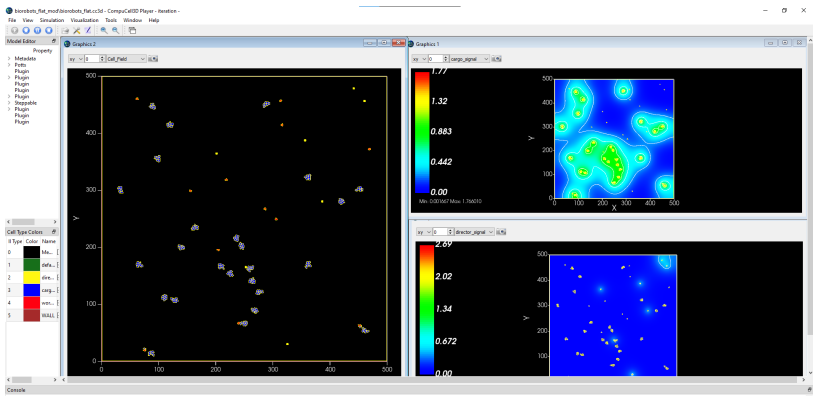
Developing automated, standardized, methods to use ABMs in multiple platforms would make re-using and validating such models cross-platform trivial. There are two ways such cross-platforms methods could work: a new universal specification standard, or model translators. The advantages of the new standard are that there is a clean slate for its development and definitions, it can be designed to be expressive, consistent and efficient, it is independent of target platforms, new platforms can adapt themselves to it, only one needs to be developed. The disadvantages are that it must be developed from scratch (there are no pre-built representations of cells, dynamics, *etc.*), until a platform adopts the standard it doesn't have a purpose, there's no pre-existing constituency of models built for it, and there are many unknowns regarding it. The advantages of translating are that it immediately has utility, and there are existing models that can be used with the translator, a target community for the translator exists. The disadvantages are that one new translator has to be made for each pair of modeling platforms, the translator has to deal with the idiosyncrasies of each platform.

To develop the translators or the universal specification several challenges need to be overcome. Some, such as how to implement universal concepts, how to deal with different scales and number of agents the platforms can simulate at once, what to do about concepts that existing only in some platforms, are common to both methods. As the universal ABM specification is more general, it will probably face more challenges and require more extensive development. To explore some of the challenges such an endeavor will encounter I've developed a translator, which is more straightforward to accomplish than a universal model specification for ABMs, to go from a PhysiCell [11] simulation to a CompuCell3D [10] simulation.

The rationale behind choosing PhysiCell and CompuCell3D is threefold: first, as mentioned, it allows for early identification and assessment of the challenges associated with developing a future standard specification; second, the cell representation and dynamics in these platforms differs greatly, making their combination a good target to uncover difficulties that would stay hidden with the translation of more similar models; third, the PhysiCell model specification is mostly done in a single XML file (as of version 1.10.4), while the CompuCell3D model specification requires distribution across three files. CompuCell3D simulations use one XML file that is responsible for the initial configuration of the simulation (simulation size, initial conditions, initial placement of cells, model component inclusion – *e.g.,* chemotaxis), one Python file containing the custom user-defined code that should be executed each time-step (which can be separated into several ”Steppable” classes), and finally a second Python file that registers the steppables classes.



(a)



(b)

Figure 4.1: Example GUI of a running simulation in: 4.1a) PhysiCell and 4.1b) CompuCell3D. Both figures show the a simulation that was translated using the methods described here: biorobots, see Section 4.7.2.

4.2 Conceptual model differences between CompuCell3D and PhysiCell models

PhysiCell and CompuCell3D have been widely utilized in the field of computational biology, each with their own distinct strengths and limitations. One fundamental difference between these frameworks lies in their representation of the simulated domain and cells. CompuCell3D's models are Cellular Potts Models (CPM) [7, 37], a lattice-based approach where the dynamics of the simulation are driven by energy minimization [7, 10], while PhysiCell uses an off-lattice approach where the dynamics are driven by forces [11]. Moreover, in CompuCell3D, cells are represented as collections of pixels or voxels in a lattice structure, whereas in PhysiCell, cells are represented as spheres with varying radii. Table 4.1 shows a few conceptual differences of CompuCell3D and PhysiCell.

This difference in representation has implications for the volume of simulation and the ability to represent complex structures. CompuCell3D simulations typically represent smaller volumes and fewer cells compared to PhysiCell simulations. However, CompuCell3D is capable of representing arbitrary shapes and complex structures that cells may form, such as renal tubules [66]. In contrast, due to the use of spherical cells in PhysiCell, representing arbitrary shapes and complex structures is more challenging.

The specification of simulations also differs significantly between the two frameworks. As mentioned, in PhysiCell most of the model definition happens in a single XML file (as of version 1.10.4). PhysiCell's XML file specification is comprehensive and allows in depth customization of model components and cell behavior. However, further model customization in PhysiCell is harder, as it requires C++ programming and lacks some helper functions for repetitive tasks. On the other hand, CompuCell3D simulations are expected to be customized using Python, and while the XML options may be less comprehensive, the framework offers greater flexibility. Both platforms have model-creation wizards that aid in model generation.

Both PhysiCell and CompuCell3D support modeling diffusing field, which can represent chemical species, virus concentrations, and other concentrations. However, the diffusion solver in PhysiCell is more advanced and faster compared to CompuCell3D. CompuCell3D uses a forward Euler solver [10], while PhysiCell uses BioFVM [11, 67]. Cells in both frameworks can interact with diffusing fields in various ways, including uptake or secretion, chemotaxis, and using the concentration as a control for cell behaviors.

One notable feature of PhysiCell is its integrated module for cell behavior simulations and changes in behaviors, known as phenotypes. They can represent the cell cycle, stages of cell death, a latent/active state of bacteria, *etc.*, see Chapter 5 for more information on phenotypes. This module is comprehensive and sophisticated, providing extensive capabilities for modeling cell behaviors in response to the microenvironment. In contrast, CompuCell3D lacks a built-in cell behavior model of this complexity and modelers are ex-

pected to create their own. The phenotype submodel in PhysiCell is complex to the point of requiring a separate project to re-implement it in a form that CompuCell3D can use.

Such an elegant representation of cell behaviors should not be restricted to one or two modeling platforms. It can be used in, and would strengthen, any bio-ABM. Therefore, I turned my Python implementation of PhysiCell's phenotypes into a stand-alone Python package called PhenoCellPy. PhenoCellPy is discussed in detail in Chapter 5 and in a separate publication [13].

Concept	CompuCell3D	PhysiCell
Cell shape	Fundamental, complex	Derived, simple
Cell shape anisotropy	Fundamental	Missing
Cell center of mass	Derived	Fundamental
Cell velocity	Derived	Fundamental
Basic cell behavior	Fundamental	Fundamental
Complex cell behavior	User-defined only	Fundamental/user-defined
Cell types	Fundamental	Fundamental
Movement bias	Derived	Fundamental
Cell compartments	Fundamental	Missing

Table 4.1: Comparison of selected concepts from CompuCell3D and PhysiCell

4.3 An Overview of the Dynamics of CompuCell3D and PhysiCell

Both platforms are physics simulators that aim to reproduce multicellular dynamics, in particular cell movement and rearrangement. In this section I'll give an overview of the strategies they employ to achieve this. In particular, it is important to note that drag forces are dominant in the cellular environment [11], this results in inertialess movement [11], *i.e.,*

$$\vec{v} \propto \vec{F} \ . \tag{4.1}$$

It is important to note that the drag in CompuCell3D is relative to the lattice itself, and that in PhysiCell the forces act on the velocity of the cell. Therefore, both platforms have drag against a fixed, universal, inertial frame (instead of, *e.g.*, against a moving fluid). This makes rigid body movement hard in both. Both platforms having a common assumption about drag, on the other hand, make the translation process of this aspect straightforward.

4.3.1 Dynamics in CompuCell3D

CompuCell3D is an implementation of the the Cellular Potts Model (*CPM*) [7, 68], also known as Graner-Glazier–Hogeweg (*GGH*) model. CPM is an modification of the Potts model [7, 69]. CPM was created to simulate multi-cellular biological systems [7]. In CPM energy minimization governs the dynamics of the system. François Graner and James Glazier developed the first iteration of Cellular Potts to test the hypothesis of cell sorting by differential adhesion [7]. Hogeweg later expanded on CPM by adding a diffusion model to CPM [68].

Graner and Glazier defined that cells are made of pixels with the same ”cell-ID,” not unlike cell in a microscopy picture are made of pixels. This means that no two cells can have the same ID, denoted by σ. In other words, regions with the same cell-ID represent a cell. In CPM cells can be of different types, and the contact energy between cells will depend on both types. This allowed Graner and Glazier to probe the differential adhesion hypothesis [7]. More energy terms are needed to fully describe cell behavior and morphology, such as an energy related to the cell volume, to chemotaxis, to cellular elongation (cells in CPM tend to be round due to surface energy minimization), directed movement, *etc*. The simplest Hamiltonian for CPM is the one defined in [7], it is

$$\mathcal{H}_{CPM} = \sum_{i} \sum_{\{j\}_i} J(\tau(\sigma_i), \tau(\sigma_j))(1 - \delta(\sigma_i, \sigma_j)) + \sum_{\sigma} \lambda_V(\sigma) \left(V(\sigma) - V_{TG}(\sigma)\right)^2 \ . \quad (4.2)$$

The first term is the contact energy and the second the volume energy. σ represent a cell (by referencing its ID), $\tau(\sigma)$ is the type of cell σ, $J(\tau(\sigma_i), \tau(\sigma_j))$ is the (type-pair dependent) contact energy between two cells, the contact energy is symmetric, *i.e.*, $J(\tau, \tau') = J(\tau', \tau)$. $V(\sigma)$ is the current volume of cell σ (in pixels), $V_{TG}(\sigma)$ is the target volume of cell σ, and $\lambda_V(\sigma)$ is the volume energy intensity for cell σ.

The double sum of the contact energy is made over a pixel i and the set of pixels inside a range, $\{j\}_i$. $\{j\}_i$ are the pixel-neighbors of i. This neighborhood is a Von Neumann neighborhood with a set order defined by the user [10]. Figure 4.2 shows how this neighborhood changes based on the range. It should be noted that using a low order neighborhood causes cells to pin to the lattice, *i.e.*, they become square and their borders are paralel to the cardinal directions of the lattice [70]. $\delta(a, b)$ is the Kronecker delta function, if $a = b$ then $\delta(a, b) = 1$ otherwise it is 0. The Kronecker delta function here guaranties that a cell will not have a contact energy with itself. The volume energy is the harmonic potential, if a cell is above or below its target volume there will be an energy penalty to the system and the cell will tent to change its volume to be closer to the target. The sum for the target volume is made over all cells.

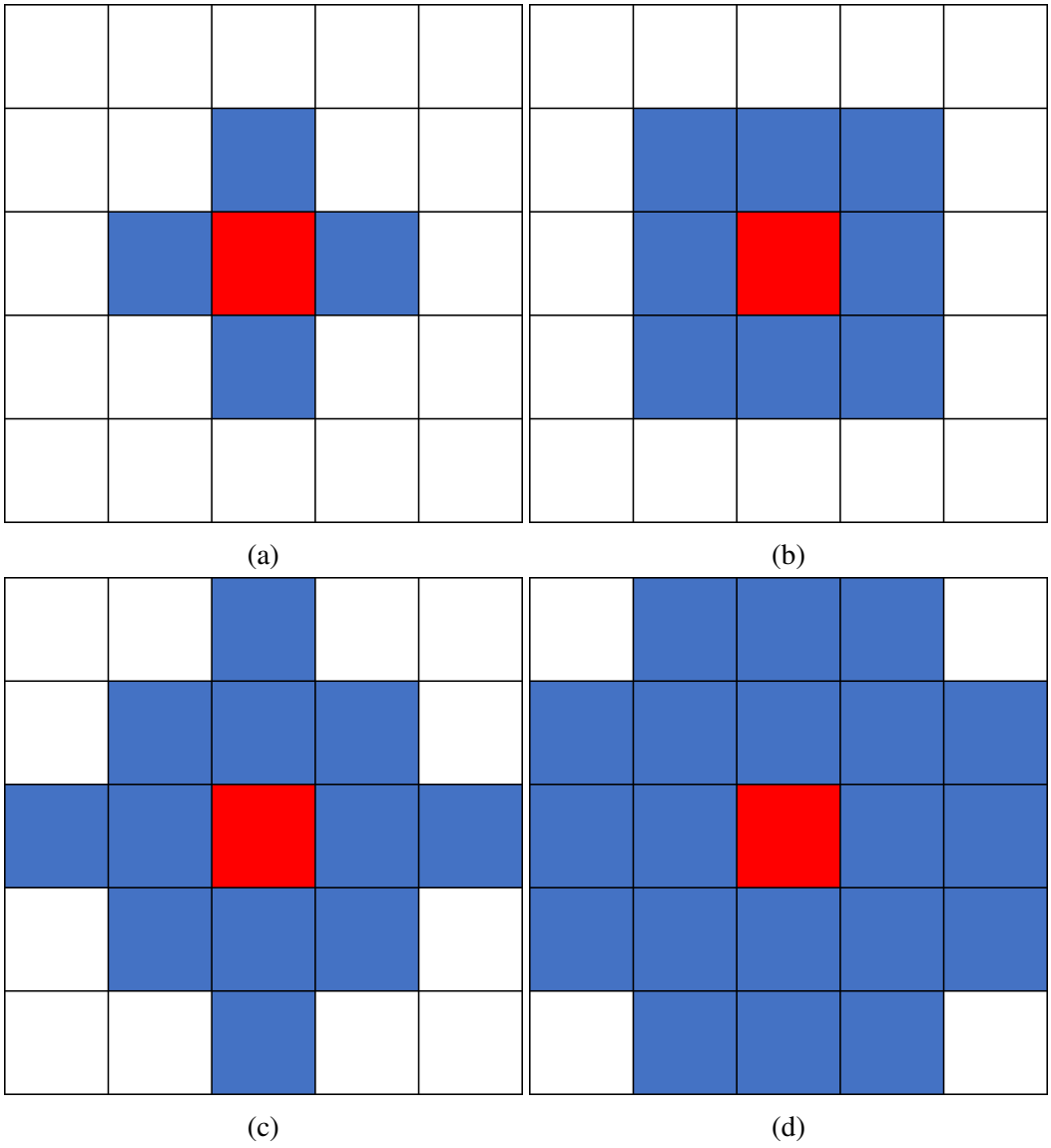


Figure 4.2: Pixel neighborhoods for each order, main pixel in red, neighborhood in blue. 4.2a) 1st order, 4.2b) 2nd order, 4.2c) 3rd order, 4.2d) 4th order.

4.3.1.1 Chemotaxis in CPM

Chemotaxis is a fundamental cell behavior that both CompuCell3D and PhysiCell represent and have available. Chemotaxis in CompuCell3D is implemented through an energy term in the Hamiltonian. In its most common form, this term is:

$$\mathcal{H}_{chemotaxis} = -\sum_{k} \lambda_k^c(\sigma) \left(c_k(j) - c_(i)\right) \ . \tag{4.3}$$

Where k is the index of the field the cell is chemotaxing on, j is the pixel the cell (σ) is moving towards and i the pixel it is moving from, c_k field k's value, λ_k^c the chemotaxing strength to that field. With a positive λ_k^c cell σ will chemotax to higher values of c_k and with a negative λ_k^c towards lower levels of c_k. If the cell doesn't chemotax then all λ_k^c are 0. See Section 4.3.2.1 for an explanation of how PhysiCell implements chemotaxis.

4.3.1.2 Temporal dynamics of Cellular Potts Model

Solving the equations for CPM analytically is impossible. Exploring all energy (*i.e.*, pixel) configurations numerically is also not doable before the heat death of the universe. The CPM algorithm chooses one pixel at random (i), then it chooses a second pixel (j) from the Von Neumann neighbors of i, it proposes that the cell that occupies i also occupies j (it proposes a pixel-ownership change), it checks what is the system's energy change due to this pixel-ownership change, and accepts or rejects the change. This has three important consequences: one as the change in energy has a direction, the model is changed from an energy formalism to a force formalism; two, updates only happen at cell borders; three, there's no detailed balance, the dynamics are not reversible (*e.g.*, if a cell disappears it can't re-appear). These dynamics correspond to biological reality, but they mean that this algorithm (a modified Metropolis algorithm [7, 37]) can't solve the equilibrium statistical mechanics that the original Metropolis algorithm was designed to solve [71, 72]. It should be noted that the background (*i.e.*, medium) cannot appear in between cells, and that the Von Neuman neighborhood order of the algorithm doesn't have to be the same as the one used by the contact energy.

The steps of the modified Metropolis algorithm used in CPM are [10]:

1. Pick a random pixel from the grid (i)
2. Pick another random pixel (j) from i's neighbors
3. If i and j don't belong to the same cell, attempt a pixel ownership change (*i.e.*, that

the cell that occupies i moves to occupy j). Think as the cell that occupies i is moving and pushing towards the cell that occupies j.

4. Calculate the change of the system's energy due to the cell movement

$$\Delta\mathcal{H} = \mathcal{H}(\text{with movement}) - \mathcal{H}(\text{without movement}) \tag{4.4}$$

5. $\Delta\mathcal{H}$ then defines a probability of accepting the move or not. It is

$$Pr(\text{accept move}) = \begin{cases} 1, \text{ if } \Delta\mathcal{H} \leq 0 \\ \exp\left(-\frac{\Delta\mathcal{H}}{T}\right), \text{ otherwise} \end{cases} \tag{4.5}$$

This process is usually called a flip-copy attempt, I call it a move-attempt. In CPM, for a grid with *N* pixels, one time-step is defined as *N* move-attempts. The usual name, for historical reasons, for the time-step in CPM is "Monte Carlo Step" (MCS). T in equation 4.5 is, also for historical reasons, called temperature. However, it has nothing to do with a temperature one would measure with a thermometer, it sets fluctuation amplitude of the cells' borders. T scales all the energies of the system, and can be set by the user. The CPM dynamics algorithm leads to a dampened movement in accordance with the equation of movement shown in Equation 4.1.

4.3.2 Dynamics in PhysiCell

PhysiCell is a center-based modelling (*CBM*) framework. In CBMs cells are represented as a particle [73] or a group of particles [74], the particle is, usually, spherical. PhysiCell uses different time-steps for calculating the solution of diffusing elements in the simulation, for calculating the cell mechanics, and for performing cell behaviors, Δt_{diff}, Δt_{mech}, and Δt_{cells} respectively. By default their values are $\Delta t_{diff} = 0.01min$, $\Delta t_{mech} = 0.1min$, and $\Delta t_{cells} = 6min$.

Unlike CPM, in center based models, forces are modeled explicitly, special consideration must be taken to respect the dominance of drag-forces. PhysiCell does this by modeling the force as acting on the cell's velocity, not on its acceleration.

4.3.2.1 Velocity in PhysiCell

PhysiCell updates each cell velocity ($\vec{v}(\sigma)$) according to the force applied to the cell, obeying Equation 4.1. The cell velocity update, in PhysiCell , occurs according to the following algorithm:

$$\vec{v}(\sigma) = \sum_{\sigma' \in \mathcal{N}(\sigma)} \left[-\vec{\mathcal{A}}(\sigma, \sigma') - \vec{\mathcal{R}}(\sigma, \sigma') \right] - \vec{\mathcal{A}}_B(\sigma) - \vec{\mathcal{R}}_B(\sigma) + \vec{v}_{mot}(\sigma) \ , \tag{4.6a}$$

$$\vec{\mathcal{A}}(\sigma, \sigma') = \sqrt{c_a(\sigma)\ c_a(\sigma')}\ \ \nabla\phi(\Delta\vec{x}(\sigma, \sigma'),\ R_A(\sigma) + R_A(\sigma')) \ , \tag{4.6b}$$

$$\vec{\mathcal{R}}(\sigma, \sigma') = \sqrt{c_r(\sigma)\ c_r(\sigma')}\ \ \nabla\psi(\Delta\vec{x}(\sigma, \sigma'),\ R_R(\sigma) + R_R(\sigma')) \ , \tag{4.6c}$$

$$\vec{\mathcal{A}}_B(\sigma) = c_{a,b}(\sigma)\ \nabla\phi(-\vec{d}(x(\sigma)),\ R_A(\sigma)) \ , \tag{4.6d}$$

$$\vec{\mathcal{R}}_B(\sigma) = c_{r,b}(\sigma)\ \nabla\psi(-\vec{d}(x(\sigma)),\ R_R(\sigma)) \ . \tag{4.6e}$$

Where σ is the cell, $\sigma' \in \mathcal{N}(\sigma)$ the cell neighbors of σ, $\vec{\mathcal{A}}(\sigma, \sigma')$ is the adhesion force between cells σ and σ', $\vec{\mathcal{R}}(\sigma, \sigma')$ the repulsion between cells σ and σ', $\vec{\mathcal{A}}_B(\sigma)$ the adhesion of cell σ to the basement membrane, $\vec{\mathcal{R}}_B(\sigma)$ the repulsion of cell σ to the basement membrane, and $\vec{v}_{mot}(\sigma)$ is the cell motility velocity (*i.e.*, persistent cell motion) [11]. $c_a(\sigma)$ and $c_{a,b}(\sigma)$ are, respectively, the cell's cell-cell and cell-basement adhesion parameters, $c_r(\sigma)$ and $c_{r,b}(\sigma)$ are, respectively, the cell's cell-cell and cell-basement repulsion parame-

ters [11]. $\Delta\vec{x}(\sigma,\sigma')$ is the distance of cell σ to cell σ' (*i.e.*, $\Delta\vec{x}(\sigma,\sigma') = \vec{x}(\sigma) - \vec{x}(\sigma')$), and $\vec{d}(x(\sigma))$ the distance from the closest point of the basement membrane to the cell [11].

$\phi(\vec{x}, R)$ is the adhesion energy potential, and $\psi(\vec{x}, R)$ the repulsion energy potential. The form of $\nabla\phi(\vec{x}, R)$ and $\nabla\psi(\vec{x}, R)$ are given in PhysiCell's supplemental materials [11], they are, respectively:

$$\nabla\phi(\vec{x}, R) = \begin{cases} \left(1 - \frac{|\vec{x}|}{R}\right) \frac{\vec{x}}{|\vec{x}|} \text{ , if } |\vec{x}| \leq R \\ 0 \text{ , otherwise} \end{cases} , \tag{4.7}$$

and,

$$\nabla\psi(\vec{x}, R) = \begin{cases} -\left(1 - \frac{|\vec{x}|}{R}\right) \frac{\vec{x}}{|\vec{x}|} \text{ , if } |\vec{x}| \leq R \\ 0 \text{ , otherwise} \end{cases} . \tag{4.8}$$

Finally, $\vec{v}_{mot}(\sigma)$ is cell σ's motility. Motility, in PhysiCell, is the persistent motion of the cell [11], it can be biased along an arbitrary direction or, more commonly, a chemical field, chemotaxis in PhysiCell is implemented through $\vec{v}_{mot}(\sigma)$. At each (mechanics) time-step a cell in PhysiCell has a probability of changing the motility direction given by

$$Pr(\text{new } \vec{v}_{mot}(\sigma)) = \Delta t_{mech}/\tau_{per}(\sigma) \ . \tag{4.9}$$

Where τ_{per} is the persistence time [11]. If a velocity update occurs, it will follow

$$\vec{v}_{mot}(\sigma) = s_{mot}(\sigma)\frac{(1-b)\,\hat{\xi} + b\vec{d}_{bias}(\sigma)}{|\,(1-b)\,\hat{\xi} + b\vec{d}_{bias}(\sigma)|} \ . \tag{4.10}$$

With $s_{mot}(\sigma)$ the cell speed, $\hat{\xi}$ a random unit vector, $\vec{d}_{bias}(\sigma)$ the bias direction vector, and b the motility randomness amount [11]. The b factor is $\in [0, 1]$, $b = 1$ represents a completely deterministic motility direction along $\vec{d}_{bias}$ and $b = 0$ Brownian motion. $\vec{d}_{bias}(\sigma)$ is determined by the cell state, chemical field interactions, or user-defined factors. If the cell

is chemotaxing then $|\vec{d}_{bias}(\sigma)| > 0$, and the value of $\vec{d}_{bias}(\sigma)$ is given by:

$$\vec{d}_{bias}(\sigma) = \sum_{j=0}^{M} \lambda_j \nabla c_j \ . \tag{4.11}$$

Where j is the index of a chemical field, ∇c_j the gradient of the field at the cell's position, and λ_j the chemotatic sensitivity to that field. If $\lambda_j > 0$ then the cell will chemotax towards higher values of c_j, if $\lambda_j < 0$ it will chemotax towards lower values of c_j. The user may opt to use a form of chemotaxis that normalizes the chemical gradient, that case ∇c_j is replaced by $\nabla c_j / |\nabla c_j|$ in Equation 4.11 [11]. If the cell doesn't chemotax then all $\lambda_j = 0$.

4.4 Implementation of the Translation Process

The PhysiCell to CompuCell3D translator works by first transforming PhysiCell's XML file into a Python dictionary using a Python package called "xmltodict" (`https://pypi.org/project/xmltodict/`). Then several functions and methods are used to parse and re-implement the available information in a CompuCell3D compliant way. After the simulation is reinterpreted the new CompuCell3D files are placed in a folder respecting the file structure CompuCell3D expects.

4.4.1 Translating Space

```
<overall>
    <max_time units="min">30240</max_time>
    <time_units>min</time_units>
    <space_units>micron</space_units>
    <dt_diffusion units="min">0.01</dt_diffusion>
    <dt_mechanics units="min">0.1</dt_mechanics>
    <dt_phenotype units="min">6</dt_phenotype>
</overall>
```

Listing 1: Example PhysiCell XML defining the time and space units and time discretization constants

```
<domain>
    <x_min>-750</x_min>
    <x_max>750</x_max>
    <y_min>-750</y_min>
    <y_max>750</y_max>
    <z_min>-750</z_min>
    <z_max>750</z_max>
    <dx>20</dx>
    <dy>20</dy>
    <dz>20</dz>
    <use_2D>false</use_2D>
</domain>
```

Listing 2: Example PhysiCell XML defining the simulated domain and space discretization constants

The first information to be extracted is the space domain size, if the simulation is 3D or 2D, and which length unit PhysiCell has set for length (micrometer, nanometer, *etc*). Listing 1 and 2 show example PhysiCell XML for defining the domain space and units respectively. It should be noted that the position of the origin within the simulation domain differs in both platforms. Specifically, PhysiCell's origin is located at the center of the domain and, as a result, has negative coordinates, while CompuCell3D's origin is positioned at a corner of the simulated domain and only has positive coordinates.

Despite being an off-lattice model, PhysiCell adopts a space discretization constant that must also be extracted. This value, along with the simulation size, is used to determine the number of pixels in the CompuCell3D simulation. Specifically, the size of each side of the CompuCell3D simulation (i.e., x, y, and z) is calculated using the following formula:

$$\Delta X_i^{cc3d} = \frac{\Delta X_i^{psc}}{\delta X^{psc}} \ . \tag{4.12}$$

Here, ΔX_i^{cc3d} represents the size (in pixels) of the i-th side of the CompuCell3D simulation, ΔX_i^{psc} indicates the size (in the corresponding units) of the i-th side of the PhysiCell simulation, and δX^{psc} denotes the discretization of space in PhysiCell. If the PhysiCell

simulation is 2D the translator will set the z dimension in CompuCell3D to be 1 pixel wide (the standard method of implementing 2D simulations in CompuCell3D). As δX^{psc} discretizes space, it must have units of [space-unit/discrete-unit]. To ensure consistency with the length units used in PhysiCell, the translator defines the pixel unit in CompuCell3D as:

$$\text{CC3D-space-unit} = \frac{\Delta X_x^{cc3d}}{\Delta X_x^{psc}} \rightarrow \left[\frac{\text{pixel}}{\text{space-unit}}\right] \ , \tag{4.13a}$$

$$\frac{\Delta X_x^{cc3d}}{\Delta X_x^{psc}} = 1/\delta X^{psc} \ , \tag{4.13b}$$

$$1\,\text{pixel} = \text{space-unit}/\delta X^{psc} \ . \tag{4.13c}$$

Appendix B.1 shows the Python function that does this process.

In CompuCell3D, a cell is represented by a group of multiple pixels, which imposes a minimum size of 1 pixel for the cell. However, it is worth noting that small cells in CompuCell3D have a tendency to fragment or completely disappear from the simulation due to their small size and energy minimization, thereby necessitating a practical minimum volume for CompuCell3D cells greater than a single pixel. Unfortunately, when converting space from PhysiCell to CompuCell3D using the methods described by Equation 4.12 and Appendix B.1, cell volumes tend to be < 1 pixel.

To respect the minimum practical size, the translator stretches the dimensions for CompuCell3D if any cell is below a minimum pixel volume (set to 8 pixels as default). This stretching process is described in Section 4.5.1.

4.4.2 Translating Time

Next, the translator extracts information about time. What are the time units used (seconds, minutes, hours, *etc*), how long the simulation should run for, and what is the discretization of time (dt). As seen in Listing 1, PhysiCell actually has three dts defined, one for the

diffusion solver, one for the mechanics solver, and one for the phenotype solver. PhysiCell uses this method to optimize simulation run time [11] by only calling each solver when it is needed.

CompuCell3D only has one time-step, the Monte-Carlo Step (MCS), which is always set to 1 [10]. To deal with diffusion instabilities, CompuCell3D sub-steps the diffusion solvers depending on the diffusion constants set. As CompuCell3D doesn't have the phenotype models it doesn't need a phenotype time-step. My PhenoCellPy [13] package (see Chapter 5) uses SciPy's ODE solvers [75] to deal with instabilities. Considering that the time-step in CompuCell3D is only equivalent to the mechanics time-step in PhysiCell, the translator only extracts it.

We want both simulations to represent the same amount of time. As dt is fixed to 1 in CompuCell3D, the number of steps the converted CompuCell3D simulation will have is:

$$\Delta\tau^{cc3d} = \Delta\tau^{psc}/dt^{psc}_{mech} \ . \tag{4.14}$$

Where $\Delta\tau^{cc3d}$ is the total of time-steps in CompuCell3D, $\Delta\tau^{psc}$ is the total amount of *time* (not time-steps) from the PhysiCell simulation, and dt^{psc}_{mech} the mechanical time-step in PhysiCell. The translator defines the time-step units in CompuCell3D (MCS's units) as:

$$\text{CC3D-time-unit} = \frac{\Delta\tau^{cc3d}}{\Delta\tau^{psc}} \rightarrow \left[\frac{\text{MCS}}{\text{time-unit}}\right] \ , \tag{4.15a}$$

$$\frac{\Delta\tau^{cc3d}}{\Delta\tau^{psc}} = 1/dt^{psc}_{mech} \ , \tag{4.15b}$$

$$1\,\text{MCS} = \text{time-unit}/dt^{psc}_{mech} \tag{4.15c}$$

Appendix B.2 shows the Python function that does this process.

This conversion leads to very high diffusion constants, which slows down Compu-

Cell3D's diffusion solvers a lot (CompuCell3D's diffusion solvers are not as sophisticated as PhysiCell's). How the translator mitigates this issue is described in Section 4.5.2.

4.4.3 Extracting Cell Types and Constraints

The present section discusses the process of extracting information about cell types and constraints in the PhysiCell-CompuCell3D translation pipeline. The first step in this process involves the extraction of cell types from PhysiCell's XML. Subsequently, the translator extracts the values associated with mechanics, custom data, and phenotype constraints. The translator converts extracted units to pixels/MCSs, using the conversion factors for time and space that were previously calculated in Equation 4.13 and 4.15. Appendix B.3 shows the functions that do the cell type and mechanics extraction.

The translator then loops over the cell constraints in the XML file, extracting the cell volume and cell-type name as the first pieces of information about the cells. If the pixel volume is below the minimum practical volume (default set to eight pixels), a flag is set to correct the conversion later in the process (described in Section 4.5.1). Following this, the mechanics constraints are extracted, including the repulsion-attraction force potential between cells, the equilibrium distance for the potential, the rate of cell adhesion molecule (CAM) [76] binding and unbinding, the spring constant for these CAMs, and the maximum distance at which two cells can start forming CAMs. It should be noted that the simulation of adhesion is complex in PhysiCell and differs from the method used in CompuCell3D, which involves defining a contact energy [10], and is much simpler. Due to this discrepancy, the translator does not attempt to implement a parameter translation for adhesion in its current version. A method to include this information is under investigation, see Section 4.5.3.

It should be noted that the default boundary condition for PhysiCell cells is periodic, to have a contained simulation PhysiCell uses an XML element, `virtual_wall_at_domain_edge`. The translation uses this element to set the bound-

ary conditions for cells in the CompuCell3D simulation. It should be noted that since, from the point of view of the simulation and cells in the simulation, nothing exists outside the simulation domain, cells near the simulation's border will experience a zero energy with the "outside." Therefore, cells tend to stick to the borders of the simulation (as the other contact energies are $J > 0$.) It is standard practice in CompuCell3D to have a wall "cell" type to build the simulation border. This way the contact with the border can be controlled. If `virtual_wall_at_domain_edge` is true, *i.e.*, the simulation's cell boundary condition is not periodic, the translator will create the wall "cell" type and build the wall for the simulation.

The phenotype data is then extracted, and the translator identifies the phenotypes that a given cell type can exhibit. For example, a cancer cell may have one phenotype while it is alive, and a different phenotype once it has died. The translator prepares the phenotype data in a format suitable for PhenoCellPy (refer to Chapter 5), and includes checks for the availability of PhenoCellPy.

As chemotaxis in PhysiCell merely biases the cell velocity while in CompuCell3D it will change the cell speed the translator, the only information the translation extracts is which cell-types chemotax on which fields and if they chemotax up the gradient or down the gradient. Lastly, custom cell parameters are extracted. These parameters are model-specific and can be related to a variety of factors. Although they are made available, the translator does not use them. Section 4.6.2 provides further details on how this information is made available.

4.4.4 Extracting Diffusing Elements

The translation of diffusing elements in CompuCell3D is performed subsequent to the correction of space conversion (as explained in Section 4.5.1), since the space correction modifies how much space each pixel represents, and, therefore, changes the diffusion constant. The process of extracting diffusing elements is similar to that of extracting cell type data.

The translator reads the corresponding XML tag, determines the number of diffusing elements, and converts the diffusion parameters. A sample XML for a diffusing element is presented in Listing 3. In Listing 3, we can see that PhysiCell sets units for the concentration of diffusing elements. These units, however, are not used for anything in PhysiCell. Therefore, the translator ignores concentration units and treats all of them as arbitrary units (AU).

```
<variable name="oxygen" units="mmHg" ID="0">
    <physical_parameter_set>
        <diffusion_coefficient units="micron^2/min">
            100000
        </diffusion_coefficient>
        <decay_rate units="1/min">0.1</decay_rate>
    </physical_parameter_set>
    <initial_condition units="mmHg">38</initial_condition>
    <Dirichlet_boundary_condition units="mmHg" enabled="true">
        38
    </Dirichlet_boundary_condition>
</variable>
```

Listing 3: Example PhysiCell XML for a diffusing element

The diffusion constant, denoted by D, has units of [space^2/time]. Considering Equations 4.13 and 4.15 and accounting for units, the conversion of D from its value specified in PhysiCell to a value that can be used in CompuCell3D is given by:

$$D^{cc3d} = D^{psc} \times \frac{dt^{psc}_{mech}}{(\delta X^{psc})^2} \ . \tag{4.16}$$

Here, D^{cc3d} is the diffusion constant for CompuCell3D with [pixel^2/MCS] as units, and D^{psc} is the diffusion constant in PhysiCell. The decay constant for diffusion has units of [1/time]. Consequently, with similar considerations, we have:

$$\gamma^{cc3d} = \gamma^{psc} \times dt^{psc}_{mech} \ . \tag{4.17}$$

Where γ^{cc3d} is the decay constant for CompuCell3D, with [1/MCS] as units, and γ^{psc} the decay constant in PhysiCell.

As mentioned, if the diffusion constants are high, CompuCell3D's diffusion solver requires extra calls for each time-step to avoid numerical errors, greatly slowing the simulation. The translator mitigates some of this issue by setting diffusing fields with high diffusion constants ($D^{cc3d} > 1000\, pixel^2/MCS$ by default) to use the steady-state diffusion solver. Using the steady-state solver is a good approximation for very high D^{cc3d}s, but not for medium levels of D^{cc3d}. To deal with medium-high D^{cc3d}s (any $D^{cc3d} \in [50, 1000)\, pixel^2/MCS$ by default) the translator will re-parameterize the time conversion (just as low cell volumes make the translator re-parameterize space), this process is described in Section 4.5.2.

Apart from converting the diffusion parameters, the translator also extracts information regarding the boundary conditions and initial conditions of diffusing fields.

4.4.5 Secretion and Uptake

This software performs the translation of secretion and uptake rates by cells of diffusing elements after it re-scales the time conversion (see Section 4.5.2), as that re-scaling would change the values of these rates.

PhysiCell has a concept of proportional-secretion [11], which CompuCell3D does not. Proportional secretion changes the secretion rate of the cell based on how much concentration of diffusing material is already near the cell. It is:

$$s_{net}(\sigma) = r_s \times (c_{tg}(\sigma) - c(\sigma)) + s_{base}(\sigma)\ . \tag{4.18}$$

Where $s_{net}(\sigma)$ is the net secretion by cell σ for a particular diffusing element, r_s is the concentration based secretion rate of that diffusing element, $c_{tg}(\sigma)$ the target concentration at which the proportional secretion will cease for cell σ, and $s_{base}(\sigma)$ an independent secretion

rate for cell σ that does not cease. It should be noted that $s_{net} \geq 0$. Even though CompuCell3D does not have a similar concept to PhysiCell's proportional secretion, it is straight forward to implement in CompuCell3Dusing simple Python operations. Section 4.6.2.2 shows how this is implemented for CompuCell3D.

Uptake, in contrast, is simple. The cells remove a fraction of the diffusing element that is near them at each time-step. *I.e.*,

$$u(\sigma) = r_u \times c(\sigma) \ . \tag{4.19}$$

It must be noted that $r_u \leq 1$ to avoid numerical errors.

As the units of secretion/uptake are proportional to $[1/time]$ and the concentration unit is not translated at all, the translation of the rates is similar to the translation of the decay rate,

$$r_i^{cc3d} = dt_{mech}^{psc} \times r_i^{psc} \ . \tag{4.20}$$

Where r_i are the secretion/uptake rates (Equation 4.18 and 4.19) or the base secretion rate (s_{base}, Equation 4.18). Appendix B.4 shows the function responsible for the secretion and uptake rates change.

4.5 Challenges

The translation process undertaken in this study has identified several challenges previously anticipated as well as revealed novel issues. While some problems such as small cells have been successfully addressed (as outlined in Section 4.5.1), the problem of translating adhesion remains unresolved. Another newly identified issue is high diffusion constants (see Section 4.4.4), the translator alleviates this issue through the implementation of a steady-state diffusion solver for diffusing fields with very high diffusion constants (see Section 4.4.4). For medium to high values of the diffusion constant, the translator re-parameterizes the time conversion process (see Section 4.5.2).

The interpretation of phenotypes, their reimplementation in Python, and improved usability and customization was a major aspect of this project. In fact, this process became so significant that it evolved into a separate project, PhenoCellPy, which is described in Chapter 5.

4.5.1 Appropriate Cell and Simulation Domain Sizes

Section 4.4.1 points out that the straight forward methods for translating space may lead to cells with < 1 pixel in CompuCell3D. To address this issue, a series of functions were implemented to stretch the simulated domain so that the smallest cell volume met the minimum requirement of 8 pixels (this minimum can be changed when running the translator). This was accomplished by expanding the cell volumes and spatial dimensions based on a calculation using the minimum allowable pixel volume,

$$r_V = \left\lceil \frac{\nu_{px}^{cc3d}}{\min\{V_{px}^{cc3d}(\sigma)\}} \right\rceil \ , \tag{4.21a}$$

$$\Delta X_i'^{\,cc3d} = r_V \times \Delta X_i^{cc3d} \ . \tag{4.21b}$$

Where r_V is the expansion ratio, $V_{px}^{cc3d}(\sigma)$ are the cell volumes in pixels, $\{V_{px}^{cc3d}(\sigma)\}$ the set of cell volumes, $\min$ the operation to get the minimum value of a set, ν_{px}^{cc3d} the minimum pixel volume the cells are allowed to have, $\lceil\ \rceil$ the ceiling (*i.e.*, round up) operation, ΔX_i^{cc3d} the previously determined simulation domain sides, and $\Delta X_i'^{\,cc3d}$ the expanded simulation domain sides. If the simulation is 2D, the translator does not change the z dimension from 1. This change implies a change to the units of the pixel (*i.e.*, how many μm/m/*etc.* it represents). Listings 4, 5, and 6 show the functions that perform this operation.

```python
def reconvert_spatial_parameters_with_minimum_cell_volume(constraints,
                                ccdims, pixel_volumes, minimum_volume):
    """
    Parameters:
    -----------
        constraints : dict
            The previously converted dictionary of cell constraints
        ccdims : tupple
            A tuple of the previously converted cc3d space parameters
        pixel_volumes : list
            A list of volumes of cells in pixels
        minimum_volume : int
            The minimum volume required for the cell in pixels
    Returns:
        - Tuple[Tuple, Dict]: A tuple containing the converted `ccdims`
            and `constraints`.
    """
    px_vols = [px for px in pixel_volumes if px is not None]
    minimum_converted_volume = min(px_vols)
    reconvert_ratio = ceil(minimum_volume / minimum_converted_volume)
    ccdims = reconvert_cc3d_dims(ccdims, reconvert_ratio)
    constraints = reconvert_cell_volume_constraints(constraints,
                            reconvert_ratio, minimum_volume)
    return ccdims, constraints
```

Listing 4: Function to reconvert spatial parameters based on the smallest cell volume, calls the functions in Listings 5, and 6. Defines what is the scaling ratio based on $\lceil \frac{\min(V_{px}^{cc3d}(\sigma))}{\nu_{px}^{cc3d}} \rceil$, where $V_{px}^{cc3d}(\sigma)$ is the list of cell volumes in pixels and ν_{px}^{cc3d} the minimum volume in pixels the cells are allowed to have.

```python
def reconvert_cc3d_dims(ccdims, ratio):
    """
    Parameters:
    -----------
        ccdims : tuple
            A tuple containing the number of pixels in each dimension
            of a 3D domain, the pixel -- real unit relationship,
            and the pixel -- real unit ratio.
        ratio : int
            The ratio to multiply the number of pixels in each dimension
            by.
    Returns:
    --------
        tuple: A new tuple containing the updated number of pixels in
               each dimension, the updated pixel -- real unit
               relationship, and the updated pixel -- real unit ratio.
    """
    ccdims = list(ccdims)
    number_pixels = [ratio * ccdims[0], ratio * ccdims[1],
                     ratio * ccdims[2]]
    old_pixel_unit_ratio = ccdims[-2]
    # pixel` = ratio*pixel = ratio * conv * unit
    new_pixel_unit_ratio = ratio * old_pixel_unit_ratio
    new_string = ccdims[3].replace(str(old_pixel_unit_ratio),
                   str(new_pixel_unit_ratio))
    new_ccdims = (number_pixels[0], number_pixels[1], number_pixels[2],
                   new_string, new_pixel_unit_ratio, ccdims[-1])
    return new_ccdims
```

Listing 5: Function to re-set the simulation domain sides (ΔX_i^{cc3d} in Equation 4.12), and pixel units (Equation 4.13), based on the ratio (r_V) defined by the function in Listing 4.

```python
def reconvert_cell_volume_constraints(con_dict, ratio, minimum_volume):
    """
    Parameters
    ----------
    con_dict : dict
       Dictionary of constraints to be converted.
       Each key in the dictionary corresponds to a type of constraint,
       and its value is a dictionary with keys like 'volume', 'surface',
       etc.
    ratio : int
       The ratio between the old voxel size and the new voxel size.
       All volume constraints are multiplied by this factor.
    minimum_volume : int
       The minimum value to replace any None values in the 'volume' key.
    Returns
    -------
    dict
       A new dictionary of constraints, where the 'volume' key values
       have been converted to the new voxel size, and any None values
       have been replaced with the `minimum_volume`.
    """
    new_con = {}
    for ctype, const in con_dict.items():
        new_con[ctype] = const
        if const['volume']["volume (pixels)"] is None:
            new_con[ctype]['volume']["volume (pixels)"] = minimum_volume
        else:
            new_con[ctype]['volume']["volume (pixels)"] = ratio * \
                                const['volume']["volume (pixels)"]
    return new_con
```

Listing 6: Function that increases the cell volumes (in pixels) based on the ratio (r_V) defined by the function in Listing 4. This function loops over the cell types in the constraint dictionary and does the multiplication. If any cell does not have a volume (in pixels) set this function will default its volume to the minimum volume ν_{px}^{cc3d}.

However, this solution presented a new problem: the expanded simulation domain became too large, with the potential to exceed 2^{64} pixels, causing a numerical overflow. Even when that hard limit is not reached, memory use becomes an issue quickly. A $1500 \times 1500 \times 1500$ lattice, for instance, would require, at a minimum, approximately 27GB of RAM due to pixel memory use. Each pixel in CompuCell3D uses 8 bytes for

each type of data shown, i.e., the cell field pixels will each use 8 bytes, then each diffusing field pixel will each also use 8 bytes, and so on.

To address this issue, the translation process truncates the simulation domain to limit the simulation size. By default, no simulation can have more than $V_{max}^{cc3d} = 3,375,000$ pixels (150^3 pixels). The function that performs the truncation process (see Listing 7) takes the previously expanded CompuCell3D dimensions (*i.e.*, the results from Equation 4.21) and the maximum simulation domain volume as arguments. If the domain size doesn't exceed the maximum, the function doesn't perform any operations. If the maximum volume is exceeded and all the sides are the same, the function sets the sides to $X_{default} = \lceil (V_{max}^{cc3d})^{1/3} \rfloor$ ($\lceil \, \rfloor$ is the operation that rounds to the nearest integer). If the sides are not the same, the translator calculates the average domain side ($\langle \Delta X \rangle$), each side's proportion to the average (P_i), and sets the new side to be the default simulation side ($X_{default}$) times the respective proportion. In other words,

$$\langle \Delta X \rangle = \frac{1}{3} \sum_{i=1}^{3} \Delta X_i'^{\,cc3d} \tag{4.22a}$$

$$P_i = \frac{\Delta X_i'^{\,cc3d}}{\langle \Delta X \rangle} \tag{4.22b}$$

$$\Delta X_i''^{\,cc3d} = \lfloor P_i \times X_{default} \rfloor \tag{4.22c}$$

Where $\Delta X_i'^{\,cc3d}$ are the increased simulation sides (see Equation 4.21b), $\lfloor \, \rfloor$ the floor (*i.e.*, round down) operator, and $\Delta X_i''^{\,cc3d}$ are the truncated simulation sides.

It's worth noting that this approach does mean that the initial conditions defined in PhysiCell can't be easily adapted, and that some simulations may not have important parts of their domain represented.

```python
def decrease_domain(ccdims, max_volume=150 ** 3):
    """
    Parameters:
    -----------
        ccdims : tuple
            tuple of 6 elements containing information about the 3D
            domain
        max_volume : int
            maximum allowed volume of the domain in pixels. Default is
            150^3 pixels.
    Returns:
        new_dims: tuple
            typle of 6 elements containing the new dimensions of the
            domain,
            after decreasing its size if necessary.
        truncated : bool
            indicates whether the domain was truncated. If True,
            a warning message is issued.
    """
    default_side = round(max_volume ** (1 / 3))
    old_dims = [ccdims[0], ccdims[1], ccdims[2]]
    old_volume = ccdims[0] * ccdims[1] * ccdims[2]
    if old_volume < max_volume:
        return ccdims, False
    if old_dims[0] == old_dims[1] == old_dims[2]:
        new_dims = [default_side, default_side, default_side]
    else:
        med_s = sum(old_dims) / len(old_dims)
        proportions = [d / med_s for d in old_dims]
        new_dims = [int(default_side * p) for p in proportions]
    new_dims.extend([ccdims[3], ccdims[4], ccdims[5]])
    message = f"WARNING: Converted dimensions of simulation domain"\
              f" totaled > {default_side}**3 pixels. \nWe have " \
              f"truncated " \
              f"the sides of the simulation.This may break the "\
              "initial conditions as defined in " \
              f"PhysiCell.\nOld dimensions:{ccdims[0:3]}\nNew "\
              f"dimensions:{new_dims[0:3]}"
    warnings.warn(message)
    return new_dims, True
```

Listing 7: Function that truncates the simulation domain sides if necessary. If the current simulation volume does not exceed the maximum volume allowed this function simply returns the dimensions as are and a flag stating that no change was performed. Otherwise, it performs the operations described in Equation 4.22, and returns the new dimensions and a flag stating that they were truncated.

4.5.2 Appropriate Diffusion Parameters

Sections 4.4.2 and 4.4.4 state that the time and diffusion parameter translations will lead to diffusion constants (D) that are very high, causing several extra calls to CompuCell3D's diffusion solvers and, consequently, a considerable simulation slow down. Some of the extra calls, and slow down, is mitigated by approximating the diffusion process with the steady-state solution, as described in Section 4.4.4.

As we need to decrease the number of extra calls to the diffusion solver we have to make the amount of time each time-step represents smaller, and proportionally increase the total number of time-steps the simulation will perform. In other words, to achieve a more general solution, the translation process reduces the time unit in CompuCell3D, *i.e.*, the amount of minutes/hours/*etc.* each MCS represent is decreased.

The translator reduces the diffusion constants in a similar fashion to the cell volume increase. By determining if any (non-steady-state) Ds are above a maximum ($D_{max} = 50 pixel^2/MCS$ by default), finding what is the ratio of D_{max} to the biggest diffusion constant, and reducing all diffusion parameters by that ratio. The translator rounds the ratio to the 2 most significant digits before the reduction process. *I.e.*,

$$r_D = \left\lceil \frac{D_{max}}{\max\{D_i^{nss}\}} \right\rfloor_2 , \tag{4.23a}$$

$$D_i' = r_D \times D_i \ . \tag{4.23b}$$

Where $\{D_i^{nss}\}$ is the set of diffusion constants (D_i^{nss}) that have not been set to use the steady-state solver, $\lceil \ \rfloor_2$ the operator to round to the 2 most significant digits, and D_i' the re-scaled diffusion constants. It's important to note that although only the non-steady-state diffusion constants (D_i^{nss}) are used to determine r_D, all diffusion constants (D_i) are re-scaled.

The other parameters that have time in their units also need re-scaling. To reduce the number of operations most of the translation happens after the re-scaling steps (*e.g.*, translation of secretion/uptake rates). Therefore, the only other parameter that needs re-scaling at this point is the decay rates (γ_i) of the diffusing elements. The process is the same as the one shown in Equation 4.23b,

$$\gamma_i' = r_D \times \gamma_i \ . \tag{4.24}$$

Where γ_i' are the re-scaled decay constants.

The function that performs the time, diffusion, and decay constants re-scaling is in Appendix B.5.

4.5.3 Cell-Cell Adhesion & Repulsion Implementation

Cell-Cell adhesion and repulsion in PhysiCell have a complex functional form (see Equations 4.7 and 4.8), CompuCell3D's contact energy can't replicate them. Even though the translator extracts all the relevant information about PhysiCell's adhesion and repulsion forces, they are not used. Instead the translator defaults to using the regular CPM adhesion energy with contact energy constants (*i.e.*, $J(\sigma, \sigma')$) set to 10, see Section 4.6.1.3.

CompuCell3D, however, has another, more complex, form of adhesion: Adhesion Flex [77]. We are currently investigating how to use adhesion flex in a way that conforms to PhysiCell's adhesion and repulsion forces, it will replace the first energy term in the Halmiltonian shown in Equation 4.2. Adhesion Flex is more flexible than the regular contact energy in a few ways. While the regular contact energy is defined in a cell-type pair basis adhesion flex can be cell on a individual cell basis. And, more importantly, it can have an arbitrary functional form. The adhesion flex contact energy is

$$\mathcal{H}_{adhesion-flex} = \sum_i \sum_{j_v} \left(-\sum_m \sum_n k_{mn} F\left(N_m(i), N_n(j)\right) \right) \left(1 - \delta(\sigma_i, \sigma_j)\right) \ . \tag{4.25}$$

The first two sums, as in Equation 4.2, are over all the pixels of the simulated domain (i), and i's neighboring pixels (j_v). The internal sums (*i.e.*, $\sum_m \sum_n k_{mn} F\left(N_m(i), N_n(j)\right)$) is the contact energy between cells σ_i and σ_j, m and n are indices for the adhesion molecules expressed by σ_i and σ_j, $F\left(N_m(i), N_n(j)\right)$ is a user-defined function of the number of molecules n and m of cells σ_i and σ_j, and k_{mn} is a scaling factor for the user-defined function F. As before, the Kronecker delta guarantees that a cell will not have a contact energy with itself.

By integrating PhysiCell's adhesion and repulsion forces and combining them together we can find an equivalent form for $F\left(N_m(i), N_n(j)\right)$, we need to be careful about how we translate the continuous maximum range from PhysiCell to a discrete range fit for CompuCell3D.

4.5.4 CPM Limitation on Cell Speed

As cells move pixel by pixel in CPM, there effectively is a maximum speed they can migrate with [78]. Typically, the center of mass of a CPM cell will not move more than $0.25 \sim 0.5$ pixel/MCS. This fact is not considered by the translator when scaling the value of the pixel and of MCS in the translated simulation, which means that cells in the translated simulation may move more slowly than they should if the original PhysiCell was to be respected. Respecting both the speed of the original cells and the original diffusion parameters is a challenge that requires future work.

4.6 Generating the CompuCell3D Simulation

After the conversion of all the relevant information from the original PhysiCell simulation to a form that can be utilized by CompuCell3D and the mitigation of any conversion issues, the files for CompuCell3D can be produced. The process of generating the files commences with the XML file. The XML file sets the cell types, diffusing fields (and associated constants and boundary conditions), and loads model plugins (*e.g.*, energies in the model's

Hamiltonian, different monitors and trackers, cell secretion and uptake). Plugins need to be loaded into CompuCell3D explicitly in order to have its capabilities available. For instance, if the volume plugin is not loaded cells can't have a target volume that they will tend towards, if the secretion plugin is not loaded the simulation can't perform secretion or uptake, and so on.

Complex behavior is defined through Python classes called "steppables" (named after "time-**steps**"). The Python file containing the steppables is the first thing generated after the XML. Finally, the translator creates and intermediary Python file that registers the steppable classes with CompuCell3D.

Upon completion of the simulation generation process, the translated CompuCell3D simulation is stored in a folder as specified by the user. In the event that no folder is specified, the translator creates a folder labeled "*CC3D_converted_sim*" in the same directory where the initial PhysiCell XML file is located and saves the translated simulation in this folder.

4.6.1 Creating the XML file

4.6.1.1 Cell Types

The translator begins the generation of the XML by creating the cell types tag. It is generated with the extraction of cell types and constraints, described in Section 4.4.3. The code that shows how this is done is in Appendix B.3. Listing 8 shows a typical cell types XML tag. CompuCell3D can set the cells to never change its constituent pixels by adding the "freeze" statement in a specific cell type tag, this is specially useful for the wall "cell".

```
<Plugin Name="CellType">
        <CellType TypeId="0" TypeName="Medium"/>
        <CellType TypeId="1" TypeName="cancer_cell"/>
        <CellType TypeId="2" TypeName="immune_cell"/>
        <CellType Freeze="" TypeId="3" TypeName="WALL"/>
</Plugin>
```

Listing 8: Typical cell type XML tag for CompuCell3D.

4.6.1.2 Simulation Domain and Boundary Conditions

Next the <*Potts*> tag is generated, this tag sets the domain size, temperature of the simulation (see [10] for a definition of temperature in Cellular Potts Models), total number of time-steps the simulation does, and if the boundary conditions for the cells is periodic. As mentioned in Section 4.3.1, the CPM algorithm will select two pixels, the second is picked from a Von Neumann neighborhood with a set range of the first. This range is defined by the "*NeighborOrder*" element in the Potts tag. This range/neighbor order is separate from the contact energy neighborhood range. Listing 9 shows a typical CompuCell3D Potts tag.

```
<Potts>
   <Dimensions x="150" y="150" z="150"/>
   <Steps>1000</Steps>
   <Temperature>10.0</Temperature>
   <NeighborOrder>1</NeighborOrder>
   <!-- <Boundary_x>Periodic</Boundary_x> -->
   <!-- <Boundary_y>Periodic</Boundary_y> -->
</Potts>
```

Listing 9: Typical Potts XML tag for CompuCell3D.

The translator adds custom tags to the Potts block regarding the relationship of pixel to real units and MCS to real units, namely <*Space_Units*>, <*Pixel_to_Space*>, <*Time_Units*>, and <*MCS_to_Time*>. They name what is used as the space unit (*e.g.*, $\mu m/nm/etc$), how much space each pixel represents, what is used as the time unit ($days\ /h/minutes/etc$),

and how much time each MCS represents, respectively. It also adds a few comments about the translation process. Listing 10 shows an example Potts tag generated by the translator. Listing 11 shows the function that generates the Potts tag and the extra information that is added by the translator.

```
<Potts>
   <!-- Basic properties of CPM (GGH) algorithm -->
   <Space_Units>1 pixel = 0.05 micron</Space_Units>
   <Pixel_to_Space units="pixel/micron" id = "pixel_to_space">10
   </Pixel_to_Space>
   <Dimensions x="150" y="150" z="1"/>
   <Time_Units>1 MCS = 10.0 min</Time_Units>
   <MCS_to_Time units="MCS/min" id = "mcs_to_time">0.02</MCS_to_Time>
   <Steps>14400</Steps>
   <!-- As the frameworks of CC3D and PhysiCell are very different -->
   <!-- PC doesn't have some concepts that CC3D does. Temperature is
   one of -->
   <!-- them, so the translation script leaves its tunning as an
   exercise-->
   <!-- for the reader -->
   <Temperature>10.0</Temperature>
   <!-- Same deal for neighbor order as for temperature-->
   <NeighborOrder>1</NeighborOrder>
   <!-- <Boundary_x>Periodic</Boundary_x> -->
   <!-- <Boundary_y>Periodic</Boundary_y> -->
</Potts>
```

Listing 10: Translator generated Potts XML tag for CompuCell3D.

```
def make_potts(pcdims, ccdims, pctime, cctime):
    """
    Parameters
    ----------
    pcdims : tuple
        Tuple of infomration from PhysiCell
    ccdims : tuple
        Tuple of infomration from  CC3D
    pctime : tuple
        Tuple with the time unit from PhysiCell
    cctime : tuple
        Tuple with the time parameters for CC3D
    Returns
    -------
    str
        Potts XML string with the given parameters.
    """
    potts_str = f"""
<Potts>
   <!-- Basic properties of CPM (GGH) algorithm -->
   <Space_Units>{ccdims[3]}</Space_Units>
   <Pixel_to_Space units="pixel/{pcdims[3]}" id = "pixel_to_space">
        {ccdims[4]}
   </Pixel_to_Space>
   <Dimensions x="{ccdims[0]}" y="{ccdims[1]}" z="{ccdims[2]}"/>
   <Time_Units>
        {cctime[1]}
   </Time_Units>
   <MCS_to_Time units="MCS/{pctime[1]}" id = "mcs_to_time">
        {cctime[2]}
   </MCS_to_Time>
   <Steps>{cctime[0]}</Steps>
   <!-- As the frameworks of CC3D and PhysiCell are very different -->
   <!-- PC doesn't have some concepts that CC3D does. Temperature is
   one of -->
   <!-- them, so the translation script leaves its tunning as an
   exercise-->
   <!-- for the reader -->
   <Temperature>10.0</Temperature>
   <!-- Same deal for neighbor order as for temperature-->
   <NeighborOrder>1</NeighborOrder>
   <!-- <Boundary_x>Periodic</Boundary_x> -->
   <!-- <Boundary_y>Periodic</Boundary_y> -->
</Potts>\n"""
    return potts_str
```

Listing 11: Function that generates the Potts tag for CompuCell3D.

4.6.1.3 Contact Energies

From Equation 4.2, the standard contact energy in CPM is:

$$\mathcal{H}_{contact} = \sum_{i} \sum_{\{j\}_i} J\left(\tau(\sigma_i), \tau(\sigma_j)\right)\left(1 - \delta(\sigma_i, \sigma_j)\right) \ . \tag{4.26}$$

Equation 4.26 implies that a CPM requires a contact energy matrix with size equal to the number of cell type pairs defined. As cells contact the medium they are in, it must also be included in the matrix. For these reasons, the translator begins by creating all the cell-type combinations (including a type with itself). After the combinations are created, the translator loops through them and generates a string that will be the contact energy XML tag. The translator defaults all contact energy to 10, and the contact energy range to 3. Listing 12 containg a typical contact enegy XML tag, and Listing 13 the translator's function that generates it.

```
<Plugin Name="Contact">
        <Energy Type1="Medium" Type2="Medium">10.0</Energy>
        <Energy Type1="Medium" Type2="cancer_cell">10.0</Energy>
        <Energy Type1="Medium" Type2="immune_cell">10.0</Energy>
        <Energy Type1="immune_cell" Type2="immune_cell">10.0</Energy>
        <Energy Type1="cancer_cell" Type2="cancer_cell">10.0</Energy>
        <Energy Type1="cancer_cell" Type2="immune_cell">10.0</Energy>
        <NeighborOrder>3</NeighborOrder>
</Plugin>
```

Listing 12: Typical contact energy XML tag for CompuCell3D.

```
def make_contact_plugin(celltypes):
    """
    Parameters:
    ----------
        celltypes : list
            A list of strings representing the cell types in the
            simulation.
    Returns:
    -------
        contact_plug : str
            A string containing the configuration for the contact plugin
    """
    combs = list(combinations(celltypes, 2))
    for t in celltypes:
        combs.append((t, t))
    combs.reverse()
    contact_plug = """
<Plugin Name="Contact">
\t<!-- PhysiCell doesn't have an equivalent to this plugin. Its  -->
\t<!-- tunning and deciding on the neighbor order is left as an -->
\t<!-- exerise to the reader. -->
\t<!-- A better option (to be implemented) is to use the adhesion flex
-->
\t<!-- Specification of adhesion energies -->
\t<Energy Type1="Medium" Type2="Medium">10.0</Energy>\n"""
    # 1 make the medium contact energies
    me = ""
    for t in celltypes:
        me += f'\t<Energy Type1="Medium" Type2="{t}">10.0</Energy>\n'
    # 2 make the combination energies
    ce = ""
    for t1, t2 in combs:
        ce += f'\t<Energy Type1="{t1}" Type2="{t2}">10.0</Energy>\n'
    contact_plug += me + ce + "\t<NeighborOrder>3</NeighborOrder>\n"\
    "</Plugin>"
    return contact_plug
```

Listing 13: Function that generates the contact energy tag for CompuCell3D.

4.6.1.4 Initial Condition

As noted in Section 4.5.1, the initial conditions from PhysiCell can't be easily adapted. Therefore the translator defaults to a simple initial condition of filling most of the simulation domain with an equal number of cells from each defined cell type placed at random.

CompuCell3D does this sort of initialization with the uniform initializer tag in the XML. Listing 14 shows a typical uniform initializer, and the translator's function that creates it is in Appendix B.5. It should be noted that the wall "cell" type is not included in this initial condition cell mix.

This tag allows to define a cuboid in which the cells are placed, all cells begin as cubes, the tag also allows to define what should be the initial side-length of the cells (*"<Width>"* in the tag).

```
<Steppable Type="UniformInitializer">
        <Region>
                <BoxMin x="10" y="10" z="10"/>
                <BoxMax x="140" y="140" z="140"/>
                <Gap>0</Gap>
                <Width>7</Width>
                <Types>cancer_cell,immune_cell</Types>
        </Region>
</Steppable>
```

Listing 14: Typical uniform initializer XML tag for CompuCell3D.

4.6.1.5 Diffusion Plugin

As the translator sets some diffusing elements to be approximated by their steady-state solution while some are kept dynamic, it need to create two Diffusion Plugins tags in the XML for CompuCell3D (one for each type of solver). The XML setting diffusion elements for CompuCell3D is complex, with elements for the diffusing element name, secretion (which is redundant with the separate diffusion plugin, and the translator does not use), and boundary conditions. The translator adds more elements to report the concentration units for that diffusion element, and several comments.

```
<Steppable Type="DiffusionSolverFE">
  <!-- Specification of PDE solvers -->
  <DiffusionField Name="oxigen">
     <DiffusionData>
        <FieldName>oxigen</FieldName>
        <GlobalDiffusionConstant>0.1</GlobalDiffusionConstant>
        <GlobalDecayConstant>1e-05</GlobalDecayConstant>
        <!-- Additional options are: -->
        <!-- <InitialConcentrationExpression>x*y
        </InitialConcentrationExpression> -->
        <!-- <ConcentrationFileName>INITIAL CONCENTRATION FIELD -
        typically a file with path Simulation/NAME_OF_THE_FILE.txt
        </ConcentrationFileName> -->
        <DiffusionCoefficient CellType="cellA">0.1
        </DiffusionCoefficient>
        <DiffusionCoefficient CellType="cellB">0.1
        </DiffusionCoefficient>
        <DecayCoefficient CellType="cellA">0.0001</DecayCoefficient>
        <DecayCoefficient CellType="cellB">0.0001</DecayCoefficient>
     </DiffusionData>
     <BoundaryConditions>
        <Plane Axis="X">
           <ConstantValue PlanePosition="Min" Value="10.0"/>
           <ConstantValue PlanePosition="Max" Value="5.0"/>
           <!-- Other options are (examples): -->
           <!-- <Periodic/> -->
           <!-- <ConstantDerivative PlanePosition="Min" Value="10.0"/>
           -->
        </Plane>
        <Plane Axis="Y">
           <ConstantDerivative PlanePosition="Min" Value="10.0"/>
           <ConstantDerivative PlanePosition="Max" Value="5.0"/>
           <!-- Other options are (examples): -->
           <!-- <Periodic/> -->
           <!-- <ConstantValue PlanePosition="Min" Value="10.0"/> -->
        </Plane>
     </BoundaryConditions>
  </DiffusionField>
</Steppable>
```

Listing 15: Typical diffusion XML tag for CompuCell3D.

CompuCell3D allows the diffusion parameters (diffusion constant and decay constant) for each diffusing element to vary on a cell-type basis [10]. The translator adds comments explaining this feature, and creates the XML elements that would define this while keeping

them commented out. CompuCell3D has more options for the diffusing elements boundary conditions than PhysiCell. In PhysiCell, they can either be Dirichlet boundary conditions (*i.e.*, with their values fixed to a constant), or "free-floating" (*i.e.*, zero derivative boundary condition). CompuCell3D is capable of simulating those, as well as non-zero derivative boundary condition, and periodic boundary conditions. Boundary conditions are set on a diffusing element basis. Listing 15 shows a typical XML tag for a diffusing element in CompuCell3D. This XML is for a non-steady-state diffusing element, the only change for a steady-state element would be in the opening XML tag, from *Type="DiffusionSolverFE"* to *Type="SteadyStateDiffusionSolver"*.

The translator's functions that generate the diffusion XML declarations are in Appendix B.7.

4.6.1.6 Secretion

As secretion from PhysiCell is complex, it must performed in Python for CompuCell3D. Therefore we only need to load it in the XML. The translator detects if there is any secretion defined and, if so, it places the XML tags from Listing 16 in CompuCell3D's XML to make the secretion plugin available. Section 4.6.2.2 shows how secretion is implemented in Python.

```
1 <Plugin Name="Secretion"/>
```

Listing 16: Secretion XML tag for CompuCell3D to load that plugin.

4.6.1.7 Chemotaxis

As chemotaxis in CompuCell3D is usually included to the cell behavior in the steppable class, the translation only adds the necessary XML to load the chemotaxis plugin, and state which fields cells will chemotax on.

```
<Plugin Name="Chemotaxis">
    <ChemicalField Name="cargo_signal"/>
    <ChemicalField Name="director_signal"/>
</Plugin>
```

Listing 17: Example chemotaxis XML tag for CompuCell3D to load that plugin and define which fields cells will chemotax on.

4.6.2 Creating Steppables File

With the XML file generated the translator begins the process of building the steppables file. Just as with the XML, the translator builds the steppables by manipulating strings. In this case of Python commands. Steppables are a special CompuCell3D Python class that can define actions to be taken at the start of the simulation, continuously during the simulation (*i.e.*, at each time-step), and at the simulation's end. This class has many utility functions, *e.g.*, to list cells of a given type, to perform cell division, to do secretion and uptake by cells, *etc*. Steppables are used to interact with cells and implement complex cell behavior. Cell parameters and attributes, such as their type, constraints (*e.g.*, target surface and target volume), custom data, can be accessed and modified in the steppables.

```
def generate_steppable(step_name, frequency, mitosis, minimal=False,
        already_imports=False, additional_init=None,
        additional_start=None, additional_step=None,
        additional_finish=None, additional_on_stop=None,
        phenocell_dir=False, user_data=""):
    imports = steppable_imports(user_data=user_data,
        phenocell_dir=phenocell_dir)
    declare = steppable_declaration(step_name, mitosis=mitosis)
    init = steppable_init(frequency, mitosis=mitosis)
    if additional_init is not None:
        init = add_to_init(init, additional_init)
    start = steppable_start()
    if additional_start is not None:
        start = add_to_start(start, additional_start)
    else:
        start += "\n\t\tpass\n"
    step = steppable_step()
    if additional_step is not None:
        step = add_to_step(step, additional_step)
    else:
        step += "\n\t\tpass\n"
    finish = steppable_finish()
    if additional_finish is not None:
        finish = add_to_finish(finish, additional_finish)
    on_stop = steppable_on_stop()
    if additional_on_stop is not None:
        on_stop = add_to_on_stop(on_stop, additional_on_stop)
    if minimal and already_imports:
        return declare+init+start+"\n"
    elif minimal:
        return imports + declare + init + start + "\n"
    elif not already_imports:
        return imports + declare + init + start + step + finish + \
            on_stop + "\n"
    return declare+init+start+step+finish+on_stop+"\n"
```

Listing 18: Generate steppable Python class master function. The later steppable-generating functions prepare each part of the steppable class and call this function to build it.

While it is possible to define the simulation in a way that a single steppable class is responsible for the whole simulation, it is usual and recommended to separate the simulation in several steppable classes. The translator separates the simulation into three steppable classes. A constraint initialization steppable, which sets cell constraints, initializes data

and parameters, and builds the wall around the simulation. A secretion-uptake steppable, responsible for performing secretion and uptake of diffusing elements. And the phenotype steppable, it is responsible for using the phenotype submodels (implemented through PhenoCellPy), and doing cell division. Although the phenotypes are used in the phenotype steppable, they are initialized by the constraint steppable.

The steppables are also capable of sharing data among them using the `shared_steppable_vars` dictionary property, and cell objects in CompuCell3D can carry their own individual data through their `dict` property [79]. The cell dictionary (*i.e.*, the cell object's `dict` attribute) is specially useful, as it allows each cell to have arbitrary information associated with it. The translator uses this to, *e.g.*, have the cell be responsible for its own secretion (see Section 4.6.2.2).

The generation of each steppable by the translator uses a helper function `generate_steppable`, Listing 18. It begins by creating the necessary imports, *i.e.*, CompuCell3D's libraries, and the phenotype model package, PhenoCellPy (see Chapter 5 and [13]). As PhenoCellPy is a new Python package, its import is done inside a try-except block, and if it fails the translator sets a flag that will bypass all of its uses in the resulting code.

Next, the function proceeds to build the steppable class name and declaration, and the steppable functions, `__init__`, `start`, `step`, `finish`, and `on_stop`, using intermediary functions (`steppable_declaration`, `steppable_init`, `steppable_start`, `steppable_step`, `steppable_finish`, `steppable_on_stop`) and information passed to `generate_steppable` by the functions that call it. Steppables in CompuCell3D can be called less often than at everystep, this period (called "frequency" by CompuCell3D) can be set ona steppable class based and is the `frequency` argument of `generate_steppable`. The intermediary functions are described in Appendix B.8.

Depending on the values of `minimal` and `already_imports`, different por-

tions of the steppable code are included in the final output. `already_imports` tells `generate_steppable` if the imports at the start of the file should be included or not. `minimal` defines if the generated steppable should contain only the `__init__` and `start` functions (`minimal = True`), or if it should also contain the `step`, `finish`, and `on_stop` functions.

4.6.2.1 Constraint Steppable

The creation of the steppables file begins with the generation of the constraint steppable. The function `generate_constraint_steppable` (Listing 19), is responsible for generating the constraint steppable. Upon invocation, this function takes several parameters, including `cell_types`, `cell_type_dicts`, `wall`, `first`, and `user_data`. The `cell_types` parameter represents the defined cell types within the simulation, while `cell_type_dicts` corresponds to the associated constraint dictionaries relevant to each cell type. The `wall` defines if a perimeter wall should be created around the simulation's domain. The `first` parameter indicates whether this function is the first steppable to be generated, in which case the initial imports are included by `generate_steppable`. Finally, `user_data` allows for optional user-defined data and is used to store the custom data extracted from PhysiCell's XML.

Internally, the function relies on auxiliary functions such as `generate_constraint_loops` (Listing 20) to iterate through the cell types and extract the respective constraint dictionaries. `generate_constraint_loops` relies on two intermediary functions that are shown in Appendix B.9.

The function `generate_constraint_steppable` also employs the `initialize_phenotypes` (Appendix B.10) function to initialize the phenotype models, producing the necessary initialization code. Chapter 5 goes into more detail of what are the phenotype initialization options. Additionally, the function constructs the required strings for wall constraints, including the sharing of steppable variables.

Ultimately, the function assembles the strings that will define the constraint steppable by invoking the `generate_steppable` function (Listing 18), passing relevant arguments such as the name, minimal flag, and any additional code snippets. The resulting constraint steppable string is returned as the output of the function.

```
def generate_constraint_steppable(cell_types, cell_type_dicts, wall,
  first=True, user_data=""):
    already_imports = not first
    loops = generate_constraint_loops(cell_types, cell_type_dicts)
    if not wall:
        wall_str = "\t\tself.shared_steppable_vars['constraints'] = self"
    else:
        wall_str = "\t\tself.build_wall(self.WALL)\n"\
                   "\t\tself.shared_steppable_vars['constraints'] = self"
    pheno_init = initialize_phenotypes(cell_type_dicts[0])
    constraint_step = generate_steppable("Constraints", 1, False,
                        minimal=True, already_imports=already_imports,
                        additional_start=pheno_init + loops + wall_str,
                        user_data=user_data)
    return constraint_step
```

Listing 19: Function that builds the constraint initialization steppable class.

```
def generate_constraint_loops(cell_types, cell_dicts):
    loops = "\n"
    for ctype in cell_types:
        this_type_dicts = get_dicts_for_type(ctype, cell_dicts)
        loop = cell_type_constraint(ctype, this_type_dicts)
        loops += loop
    return loops
```

Listing 20: Function that builds the start function loops for the constraint initialization steppable. Each defined cell type has a loop over all cells of that type that initializes the constraits relevant to that cell type.

4.6.2.2 Secretion and Uptake Steppable

The generation of the secretion and uptake steppable is handled by the `generate_secretion_uptake_step` function (see Listing 21). As usual, behaviors are defined on a cell-type basis (and can be changed on a cell basis), therefore this

function takes as an argument the list of cell types. It also takes the dictionary containing secretion-uptake information (`sec_dict`), the steppable call period (`secretion_dt` , the "frequency" in the steppable class), and a flag stating if this will be the first steppable class in the file. If the secretion dictionary is empty we don't need to have a secretion-uptake steppable, therefore the function exits early returning an empty string.

```
def generate_secretion_uptake_step(cell_types, sec_dict,
  secretion_dt=None, first=False):
    if not sec_dict:
        message = "WARNING: no secretion data found\n"
        warnings.warn(message)
        return ''
    if secretion_dt is None:
        secretion_dt = 1
    already_imports = not first
    field_names = get_field_names(sec_dict)
    secretors = make_secretors(field_names)
    loops = make_secretion_uptake_loops(cell_types, sec_dict)
    sec_step = generate_steppable("SecretionUptake", secretion_dt,
                    False, already_imports=already_imports,
                    additional_start=secretors, additional_step=loops)
    return sec_step
```

Listing 21: Secretion-Uptake steppable class generator function. It exits early if there is no secretion data, if a steppable period is not defined it defaults to setting the steppable period to one (*i.e.*, this steppable runs at every time-step in CompuCell3D). It then calls helper functions to build the steppable `start` and `step` functions, and passes the strings to `generate_steppable` . Finally it returns the steppable as a string.

To do secretion or uptake CompuCell3D uses a specific object to interact with the fields that has to be initialized, the "secretor" object. The secretion-uptake steppable generation function extracts the names of the diffusing elements that have secretion or uptake from the secretion data dictionary (using `get_field_names`). With those names it generates CompuCell3D code that will create the secretor objects in CompuCell3D and have them saved in a dictionary for later use during the simulation steps (see Listing 22).

```python
def make_secretors(field_names):
    sec_list = "\t\tself.secretors = {"
    for name in field_names:
        sec_list += f"'{name}': self.get_field_secretor('{name}'),"
    sec_list = sec_list[:-1]
    sec_list += "}\n"
    return sec_list
```

Listing 22: Function that generates the secretor objects declaration for CompuCell3D. Secretors in CompuCell3D are created using the `self.get_field_secretor(FIELD_NAME)` function, `make_secretors` loops over the field names and creates a string that will generate the secretors and save them to a dictionary in CompuCell3D. The keys of that dictionary are the field names and the item of the key is the secretor.

The secretion-uptake generation function then creates the CompuCell3D loops that will have the secretion and uptake calls for each cell (see Listing 23 and 24). The `make_secretion_uptake_loops` (see Listing 23) function generates a string representation of secretion and uptake loops for all cell types. It takes the cell types, secretion dictionary, secretors dictionary, and field names as inputs. The function initializes a loop structure for iterating over the secretors using the `self.secretors` dictionary. It then iterates over each cell type and generates secretion loops only for cell types present in the secretion dictionary. It extracts the comment from the secretion dictionary and calls the `make_secretion_loop` (Listing 24) function to generate the secretion loop string. Finally, it concatenates the secretor loop string with the generated secretion loops and returns the resulting string.

The `make_secretion_loop` function generates a string representation of a secretion loop for a specific cell type. It takes the cell type and a comment as input. The function defines the loop structure using the `generate_cell_type_loop` helper function. It includes code to check if the field name exists in the cell's dictionary and retrieves the corresponding data. It calculates the net secretion based on the secretion rate, secretion target, and amount seen by the cell, as described in Section 4.4.5 and Equation 4.18. In

more detail, the secretion and uptake parameters are saved to the cell dictionary bu the constaint steppable. Then the secretion steppable will read those parameters, calculate the net secretion and call the secretor's secretion and uptake functions. The secretion and uptake rates are saved to dictionary entries: "secretion_rate," "secretion_target," "uptake_rate," "net_export," "secretion_rate_MCS," "net_export_MCS," "uptake_rate_MCS."

The translator includes explanatory comments regarding the different secretion locations in CompuCell3D and the implementation of PhysiCell's net-secretion functionality with the generated steppable. Finally, it calls the secretion and uptake methods of the secretor object. The function returns the generated secretion loop string.

```
def make_secretion_uptake_loops(cell_types, sec_dict):
    secretor_loop = \
        "\t\tfor field_name, secretor in self.secretors.items():\n"
    loops = ""
    for ctype in cell_types:
        if ctype in sec_dict.keys():
            comment = \
                sec_dict[ctype][
                    list(sec_dict[ctype].keys())[0]]\
                    ['secretion_comment'] +'\n'
            loops += make_secretion_uptake_loop(ctype, comment)
    return secretor_loop + loops
```

Listing 23: This function loops the cell types and diffusing elements that are involved in secretion or uptake and creates a CompuCell3D loop over cells for each cell type and secretor object.

```
def make_secretion_uptake_loop(ctype, comment):
    # secretion in physicell is
    # secretion rate * (target amount - amount at cell) + net secretion
    # looking at units that is correct:
    # < secretion_rate units = "1/min" > 0 < / secretion_rate >
    # < secretion_target units = "substrate density" > 1
    #< / secretion_target >
    # < uptake_rate units = "1/min" > 0 < / uptake_rate >
    # < net_export_rate units = "total substrate/min" > 0
    #< / net_export_rate >
    loop = generate_cell_type_loop(ctype, 3)
    check_field = "\t\t\t\tif field_name in cell.dict.keys():\n"\
        "\t\t\t\t\tdata=cell.dict[field_name]\n"
    seen = "\t\t\t\t\tseen = secretor.amountSeenByCell(cell)\n"
    secrete_rate = "\t\t\t\t\tnet_secretion = max(0, "\
        "data['secretion_rate_MCS']"\
        " * (data['secretion_target'] - seen)) +"\
        " data['net_export_MCS']\n"
    where_secrete = "\t\t\t\t\t# In PhysiCell cells are point-like,"\
        " in CC3D they"\
        " have an arbitrary shape. With this " \
        "\n\t\t\t\t\t# CC3D allows several different secretion "\
        "locations: over"\
        "the whole cell (what the translator uses),\n\t\t\t\t\t#"\
        " just inside the "\
        "cell surface, just outside the surface, at the surface."\
        " You should "\
        "explore the options\n"
    secrete = "\t\t\t\t\tsecretor.secreteInsideCell(cell,"\
        " net_secretion)\n"
    uptake = "\t\t\t\t\tsecretor.uptakeInsideCell(cell, 1e10, "\
        "data['uptake_rate'])\n"
    return loop + check_field + seen + comment + secrete_rate + \
        where_secrete + secrete + uptake
```

Listing 24: This function creates the loop over cells of a given type. It adds the calculation for net secretion for a given cell by fetching how much of the diffusing element the cell is exposed to and cell specific data for that diffusing element. It then concatenates all parts of the loops and returns them.

Now that the secretion-uptake steppable generator function has built all the necessary strings that will perform secretion and uptake in CompuCell3D, it uses `generate_steppable` to assemble the steppable.

4.6.2.3 Phenotype Steppable

The bulk of the necessary steps to use PhenoCellPy's [13] phenotypes is done at their initialization. This, as mentioned, is done by the constraint steppable. Therefore, the phenotypes steppable is only responsible for time-stepping the phenotypes, monitoring their return values, and performing cell division.

The `generate_phenotypes_loops` (see Listing 25), with a helper function (see Listing 26), creates the loops that will do the phenotype time-step. At each time-step a cell using PhenoCellPy can change its volume, the translator adds code to change the CompuCell3D's cell volume consistently. As volumes in PhenoCellPy have real units and volumes in CompuCell3D are expressed in pixels, the translator uses the conversion factor determined in Section 4.4.1 to keep both volumes consistent.

```
def generate_phenotype_steppable(cell_types, cell_dicts, first=False):
    already_imports = not first
    loops = generate_phenotypes_loops(cell_types, cell_dicts)
    pheno_step = generate_steppable("Phenotype", 1, True,
                        already_imports=already_imports,
                        additional_step=loops)
    return pheno_step
```

Listing 25: Phenotype steppable generator function. It calls `generate_phenotypes_loops` to generate the loops that update the phenotype models.

```
def generate_phenotypes_loops(cell_types, cell_dicts):
    loops = "\n"
    loops += "\t\tcells_to_divide = []\n\t\tif pcp_imp:\n\t\t\tpass\n"
    for ctype in cell_types:
        this_type_dicts = get_dicts_for_type(ctype, cell_dicts)
        loop = type_phenotype_step(ctype, this_type_dicts)
        loops += loop
    loops += f"\t\t\tfor cell in cells_to_divide:\n\t\t\t\t"\
             "# WARNING: As cells in CC3D have shape, they can be " \
             f"divided along their minor/major axis, randomly in" \
             " half, or along a specific vector\n"
    loops += "\t\t\t\tself.divide_cell_random_orientation(cell)\n"
    loops += "\t\t\t\t# self.divide_cell_orientation_vector_based"\
        "(cell, 1, 1, 0)\n"
    loops += "\t\t\t\t# self.divide_cell_along_major_axis(cell)\n"
    loops += "\t\t\t\t# self.divide_cell_along_minor_axis(cell)\n"
    return loops
```

Listing 26: Function that generates the phenotype update loops. Each cell type has its own loop to perform this update. It also creates code to append cells that divide to a list, to loop that list, and call the division method on those cells.

At each time-step the phenotype of the cell may flag that the cell has changed phases (see Chapter 5 for the definition of phenotype phase), should be removed from the simulation (*e.g.*, because it has died, or migrated), or that it has undergone cell division (either mitosis or meiosis) [13], see Chapter 5. The translator implements the standard method of performing cell division in CompuCell3D: dividing cells are appended to a separate list, and the program loops over that list and calls the division function on each cell.

The translator leaves the implementation of behaviors related to phase change and cell removal as a task the user should do (this is pointed out by comments in the generated steppable). The translator doesn't implement any specific behaviors for this case for a few reasons. The main reason for not implementing cell removal is that, when using a Cellular Potts Model, a cluster of cells may misbehave if a cell is outright removed from its center during a single time-step. Doing so is equivalent to creating a vacuum and can be disruptive to the simulation. If cells are isolated, on the other hand, this poses no issues. A reason for not performing any action on phase change is that those are very model specific and the

translator wouldn't be able to define a best general approximation.

PhenoCellPy will change internal behaviors (what volume the cell should have, how fast can it change its volume, should the cell calcify, *etc*) automatically. Therefore, it is not, usually, necessary to implement extra behaviors in that case. The user may still want to add statistics related to phase change, or change other aspects of the simulation or the cell at that point.

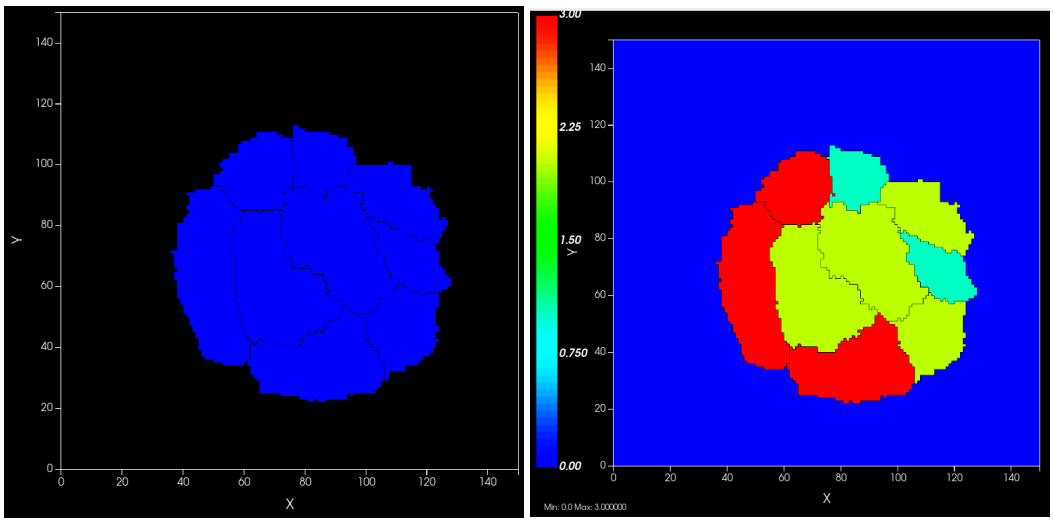
4.7 Results (example translations)

We chose two PhysiCell simulations to translate as proof of concept. We choose these two simulations for a few reasons, they are available to run online on nanoHUB, they exemplify important PhenoCellPy concepts, and the level of extra work that has two be done after the automated translation process is minimal for one of them and extensive for the other.

4.7.1 Cell Cycle

The cell cycle example shows how the cell changes volume during PhysiCell's "Flow Cytometry Cell Cycle," it is available to run online [80]: `https://nanohub.org/tools/trcycle`. It is a very simple simulation, it starts with a single cell placed in the center of the lattice, and monitors its division. The online version has a few configuration options, we opted to translate the simplest version. It's important to note that this simulation doesn't specify what units it is using nor what should be its volume (in this case PhysiCell uses a default volume). Therefore, the translated version also doesn't have units, and the pixel cell volume is set to default minimum.

The purpose of this simulation is to show the progression of phases of a PhysiCell phenotype and how the cell volume changes during the whole cycle. As phenotypes are key, PhenoCellPy [13] (see Chapter 5) is necessary to run it.



(a) (b)

Figure 4.3: Translated cell cycle simulation. 4.3a) The cells in the simulation. 4.3b) Color coded cells based on which phase of the cycle they are in.

This simulation requires minimal extra work. We increased the cell volume and the simulation domain size to make visualization easier and added trackers for cell volume and phenotype phase.

Both the original translation and the modified translation are available in the translator's github page: `https://github.com/JulianoGianlupi/pcxml2cc3d/tree/9f6c8e7ffc927cd46bbe76f7b56b49d95bbdec41/example-translations/cell_cycle`.

4.7.2 Biorobots

The biorobots [81] simulation does not rely on PhenoCellPy's phenotypes, it's goal is to show mechanics and how simple rules applied to agents can lead to complex behavior. This simulation has three cell types: director, worker, and cargo. The directors and cargo are immobile and each secrete a different chemo-attractant. The workers' goal is to place the cargo with the director cells, to do so they seek cargo (using the cargo's chemo-attractant) and attach to the cargo. Then they switch their chemo-attraction from the cargo chemo-attractant to the directors' chemo-attractant. Once they found the director cell they release

the cargo and switch their chemo-attraction back to be cargo-seeking. Once a cargo cell attaches to a worker cell it stops secreting the chemo-attractant [81]. PhenoCellPy's biorobots simulation is available to run online `https://nanohub.org/tools/pc4biorobots`, the biorobot's landing page has a more detailed description of the simulation.

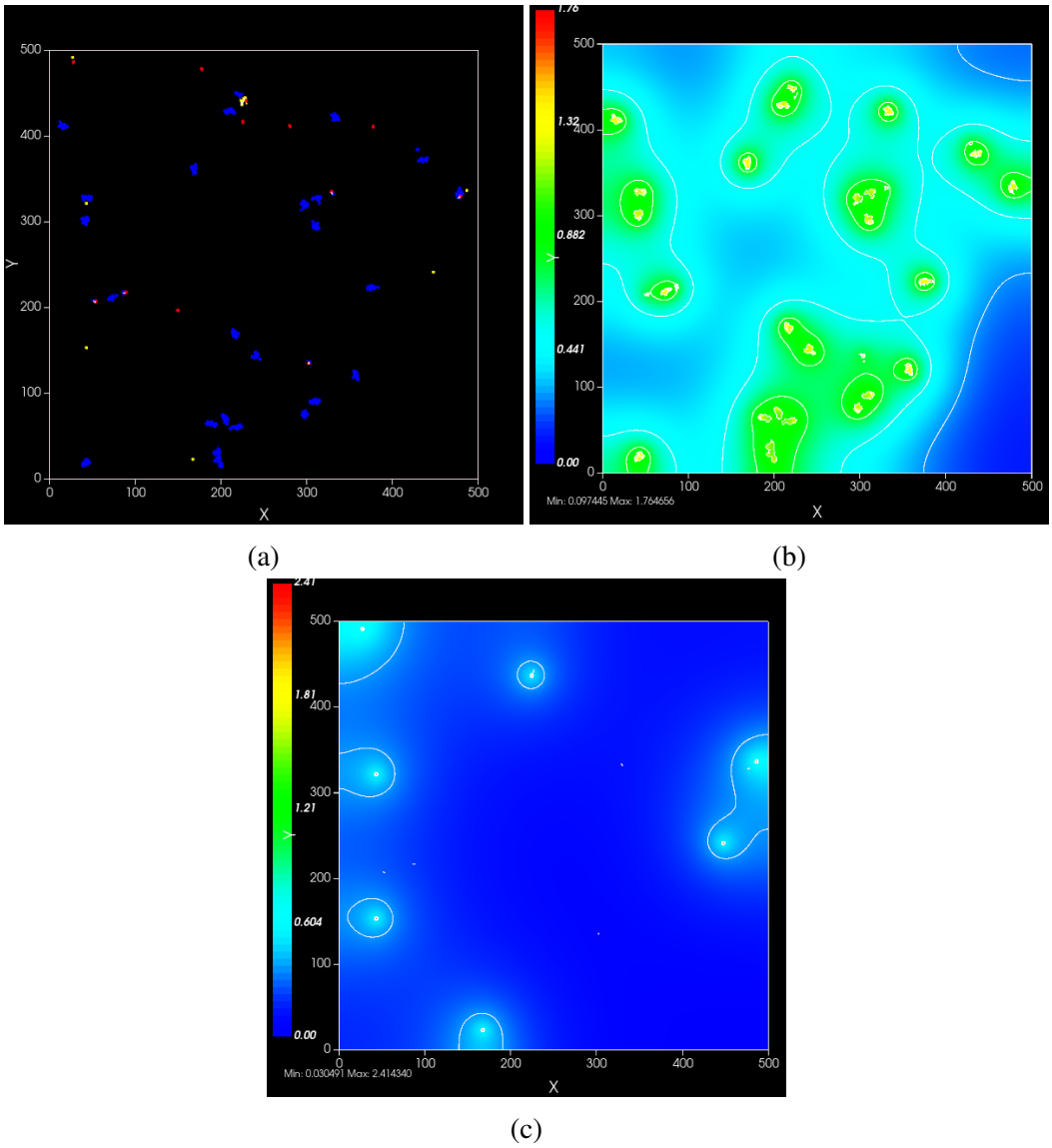


Figure 4.4: Translated biorobots simulation. 4.4a) The color-coded cells (cargo in blue, workers in red, and directors in yellow). 4.4b) The cargo cells' chemo-attractant field levels. 4.4c) The directors' chemo-attractant field levels.

This simulation requires more extensive post-translation work when compared to the cell cycle simulation. It has several behavioral rules that need to be applied, the initial cell

placement is complex, and it requires some CompuCell3D plugins to be included that are not automatically generated by the translator.

The extra steps needed to finish the biorobots translation are:

- Load the extra plugins: Neighbor Tracker, Focal Point Plasticity, External Potential. Neighbor Tracker is used to detect cell neighbors and manipulate them, and Focal Point Plasticity and External potential help the worker cells carry their cargo. Focal Point Plasticity creates a spring-like energy potential between two cells, External Potential allows for the use of arbitrary forces on cells.

- Remove the default cell placement and re-create the initial placement from PhysiCell. Cargo cells are placed in groups of seven, each group is placed in a random empty space of the simulation. Worker and director cells are placed randomly.

- Re-implement the agent rules from the PhysiCell simulation. *E.g.*, switching the chemo-attraction of worker cells from one field to the other.

For practicality purposes, the director cells set to be frozen, and when a cargo cell is delivered the simulation transforms that cargo cell into a director cell (but the previously cargo cell does not secrete the director chemo-attractant). The cargo attachment to the worker is done by creating a focal point plasticity link between the work and cargo cells and, to help the workers pull their cargo, the simulation applies a force to the cargo cell thqat points towards the center of the worker cell.

The translated simulation time to completion is orders of magnitude higher when compared to the original. The online deployment of the original, running online on limited resources, finishes the run in 2 minutes. The translated version, running on a Windows PC with 16 GB of RAM took over 24 hours to finish. This extreme time to completion was measured *after* we reduced the simulated domain of the translated simulation from $1837 \times 1837\ pixels$ to $500 \times 500\ pixels$.

Both the original translation and the modified translation are available in the translator's github page `https://github.com/JulianoGianlupi/pcxml2cc3d` under the directory "example-translations/biorobots".

4.8 Discussion

The development and implementation of the translator for converting PhysiCell simulations to CompuCell3D format is a crucial step in facilitating cross-platform compatibility and enabling the utilization of simulation models in different computational frameworks. It successfully converts PhysiCell simulations to CompuCell3D format, demonstrating the feasibility of converting model definitions from diverse simulation frameworks (Cellular Potts and center-based). By converting essential information such as simulation parameters, cell types, diffusion and decay constants, and model plugins, the translator ensures compatibility between the two frameworks. In this study, we presented a detailed description and analysis of the translator, highlighting its key functionalities, challenges encountered during the conversion process, and solutions to some of the challenges.

The XML file structure employed by PhysiCell simplifies the translation process by centralizing and easily parsing model information (as of version 1.10.4). This highlights the potential of XML or similar formats as effective means of general agent-based model specification.

The successful implementation of the translator offers researchers a valuable tool for transitioning their PhysiCell simulations to the CompuCell3D framework. By providing a means to overcome platform-specific incompatibilities and enabling simulation interoperability, the translator enhances collaboration and facilitates the exploration of diverse computational biology models and methodologies.

While the translation process may require some additional implementation steps left to the user (see Section 4.8.1), the translator performs the majority of the work and generates a functional simulation. It demonstrates the feasibility and value of creating a general

agent-based model by successfully translating model definitions across different frameworks. Moreover, it proves that the endeavor of creating a general agent-based model can be fruitful and is worthwhile, as it is possible to go from one model definition to a very different style of definition and model. Therefore, the creation of a general definition is doable.

We also present what are the likely points of difficulty, namely concepts that are missing from one platform (*e.g.*, phenotypes in CompuCell3D), differences in simulation scales (*e.g.*, simulation size, and other platform specific limitations), or the differences in the dynamics (*e.g.*, adhesion force in PhysiCell and contact energy in CompuCell3D). The implementation of the translator shows possible solutions to those difficulties, such as creating a general implementation of missing components, like the creation of PhenoCellPy [13], see Chapter 5.

As a proof of concept and prototype the translator we developed and presented is successful in doing the bulk of the translation work for the modeler.

4.8.1 Limitations & Future Work

The translator for converting PhysiCell simulations to CompuCell3D format has certain limitations that should be acknowledged. These limitations arise from the differences between the two frameworks and the challenges involved in translating simulation models across platforms. Some of these limitations were expected and some were identified by the exercise of building the translator. All of these will most likely also be relevant to the creation of a universal ABM model specification formalism.

Incomplete Translation: The translation process performed by the translator is not fully comprehensive and may leave certain implementation steps to be addressed by the user. For instance, the initial placement of cells is not converted, and a few model components need to be calibrated by hand.

Platform-Specific Concepts: The translator encounters difficulties when dealing with

concepts that are specific to one platform but absent in the other. For example, CompuCell3D lacks the notion of phenotypes present in PhysiCell. The creation of PhenoCellPy has addressed this particular limitation, but it involved a spin-off project. Another example is the translation of cell adhesion and repulsion, which is not done. We are currently investigating a method of using an alternative energy term in CompuCell3D for cell contact as a method to implement PhysiCell's adhesion and repulsion forces (see Section 4.5.3). Addressing these differences and finding suitable equivalents or workarounds is essential for a successful translation and a successful universal model specification standard.

Scale and Limitations: Differences in simulation scales, such as total simulation domain size, the time-scale of a time-step, the length-scale of the simulation, and number of agents. The computing requirements of a platform can limit what can be ported from the other platform. We have performed several adjustments to simulation size, simulation spatial length (*i.e.*, how many total μm the simulation covers), simulation time-scale, and other parameters to have a simulation that is capable to run.

Dynamics and Behavior: The dynamics and behavior of the simulation models may differ between PhysiCell and CompuCell3D (or any other two platforms) due to variations in underlying algorithms and modeling approaches. Adapting these dynamics to the target platform may require careful consideration and potential adjustments.

User Responsibility: The translator performs the majority of the work in the translation process but places some responsibilities on the user. Implementing missing components or addressing specific platform requirements may require additional effort and expertise from the user.

Run Time Differences: PhysiCell style simulations are generally faster to run. In CPM, at each time-step, the dynamics need at least "number of pixel" calculations of the energy change to get the pixel move probability (see Section 4.3.1, and Equations 4.2 and 4.5). As more energy terms are added this calculation becomes more costly. PhysiCell in particular uses a very well optimized diffusion solver, whereas CompuCell3D uses a simple forward

Euler solver.

These limitations highlight the complexities and nuances involved in translating simulation models between different computational frameworks and in building a universal model specification for ABMs that will work for all platforms and methodologies.

4.8.2 Software requirements

The translator itself is very lightweight and runs in seconds, posing no requirements for hardware. The only requirements are Python packages that it uses and are not part of Python's standard library. They are:

- `xmltodict`, `https://github.com/martinblech/xmltodict`,
- `autopep8`, `https://github.com/hhatto/autopep8` (optional).

CHAPTER 5
PHENOCELLPY: A PYTHON PACKAGE FOR BIOLOGICAL CELL BEHAVIOR MODELING

5.1 Introduction

Repeatedly in biology we see stereotyped sequences of transitions between relatively distinct phases. While such sequences of phases occur at the scale of societies, ecosystems, animals, organelles, macro-molecular machines, the classical example occurs at the scale of cells. This paper will primarily discuss sequences of cell phases, but the concepts and tools developed generalize easily to other biological agents.

A typical example for cells would be the cell cycle, either simplified as proliferation followed by quiescence (see Section 5.3.2), or in its more complete form of G0/G1 followed by S, G2, M (Figure 5.1a, see Sections 5.3.4 and 5.3.5). Another example would be a model of a SARS-CoV-2 infected cell, the cell transitions from an uninfected state (or phase) to an infected state with no viral release (eclipse phase), then to a virus releasing state, and finally dies [1] (Figure 5.1b).

For biological systems modeling, it is important to have a structured definition of biological states, sequences of states, and conditions to go from one state to the next. Abstracting cellular behaviors, or, more generally, biological-agent behavior, into a computer model requires the modeler to simultaneously understand the biology of interest and the implementation of that biology in a computational framework. To make modeling more accessible to biologists we need to simplify the transition from the biological description to computational implementation.

There are many computational frameworks for modeling multiscale cellular system: CompuCell3D, Tissue Forge, Morpheus, PhysiCell, Artistoo, FLAME, *etc.* Currently, each

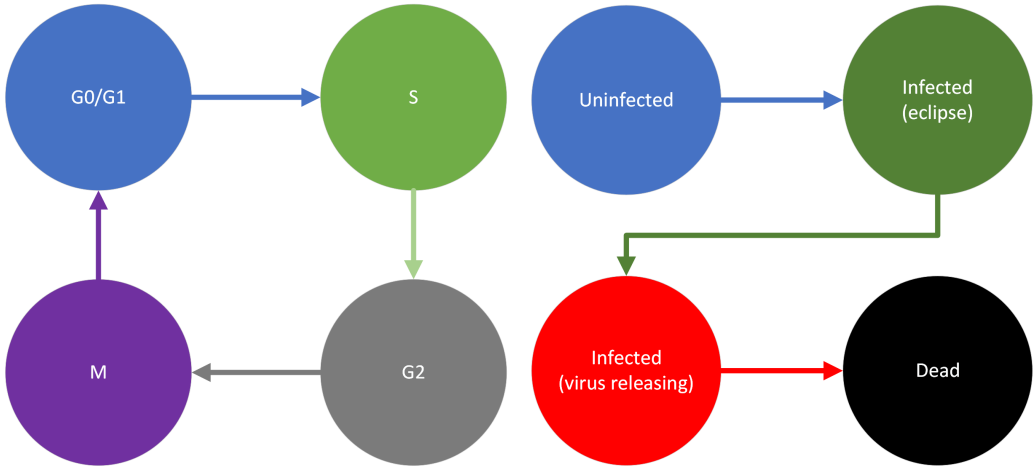


Figure 5.1: Example sequences of cell behaviors. a) Cell cycle. b) Stages of viral infection of a cell.

modeling framework requires a different type of description of behaviors and behavior transition triggers, and often do not have a defined standard for basic biological *processes* like "cell cycle". Often implementations of cell, or biological-agent, behaviors are platform-specific and even modeler-specific. This results in models of cell behaviors that are difficult to interpret, share and re-use as well as being restricted to the platform they were created in. This situation makes it difficult to separate the biology being modeled from the framework it is being modelled in. Ideally, the description of the biology being modeled should be separate from its algorithmic implementation.

For instance, a model of sequence of cell behaviors built for CompuCell3D [10] of, *e.g.*, infected cell states [1], can't be simply copied and re-used as-is in some other platform (*e.g.*, Tissue Forge [82]). The modeler has to back out the underlying conceptual model of the phases and their transition rules by interpreting the original implementation, which mixes the biological concepts and the implementing code, and re-code it according to the model specification structure in the new platform. This interpretation is time consuming and subject to many types of error, including misunderstanding of the original model structure and transition rules, mismapping of model-specific parameter values and

the possibility of introducing coding errors when the conceptual rules are remapped to code in the new model specification language.

The PhysiCell [11] package has developed a really elegant way to do this. Phenotypes[1] are defined as the sequence of behaviors and are implemented as a class. The different behaviors that make up a phenotype are called phase. The trigger to go from one phase to the next can be set to either be deterministic after a set time, or stochastic with a set rate. The transition can also depend on environmental factors, or on the cell size. We believe PhysiCell's approach to phenotypes would be valuable in many other multicellular model frameworks, *e.g.*, in CompuCell3D [10], and Tissue Forge. Currently PhysiCell's implementation of phenotypes is in C++ and is closely linked with the PhysiCell package, making it difficult to reuse in other modeling frameworks.

PhenoCellPy implements and makes available for general use PhysiCell's phenotype functionality in in the form of a Python package. It also makes the process of generating new phenotypes, phases, and phase change triggers easy. This addresses the issues of lack of standards and platform specificity, by creating an easy to use and platform-independent Python package that can be embedded in other models. PhenoCellPy is an open-source package, its source code is available at its GitHub repository [83]. Although the concepts and methods of PhenoCellPy are general to many types of biological agents (cells, mitochondria, nucleus, certain organelles) it was built with the cell as the focus.

Phenotype here can mean the cell cycle, the sequential stages of necrosis, the fact that the cell is alive, the fact that it is dead, the different stages of viral infection, *etc*. Phenotype is, then, the set of observable characteristics or traits of an organism.

In PhenoCellPy we create methods and Python classes to define cell behaviors, sequences of cell behaviors, and rules for the behavior switching. We have built several pre-packaged models representing phenotypes of cell behaviors to show-case PhenoCellPy's capabilities (see Section 5.3).

[1]Nomenclature in biology is diverse, with several different definitions for the same term. We define what phenotype and phase mean in the context of PhysiCell and PhenoCellPy in our text.

An abstract Phenotype consists of one or more Phases (Section 5.2.2), each Phase defines the volume of the cell, and the volume change rates the Phase displays. It also defines which is the next Phase of the Phenotype, what conditions trigger Phase change, if the agent should divide when exiting the Phase, what behaviors occur immediately on Phase entry and just before Phase exit (*e.g.*, changing the target volume). The cell volume dynamics are handled by the Cell Volume class. The cell volume is subdivided among the solid and fluid cytoplasm, solid and fluid nucleus, and a calcified fraction.

```
# making an empty list to save the diving cells to
dividing_cells = []
# looping over all CompuCell3D cells
for cell in self.cell_list:
    # calling PhenoCellPy's time-step and saving the flags returned
    # from it.
    # each cell has its own cell dictionary (cell.dict), we have
    # initialized each
    # cell's phenotype to cell.dict["phenotype"]
    changed_phase, should_be_removed, divides = \
            cell.dict["phenotype"].time_step_phenotype()
    if changed_phase:
        # if the phenotype of cell changes phase we call extra tasks
        # we may have defined on the agent
        self.phase_change_tasks(cell)
    if should_be_removed:
        # if the phenotype returns that the cell should be removed from
        # the simulation we call CompuCell3D's deletion method
        self.delete_cell(cell)
    if divides:
        # if the phenotype of cell says the cell has divided we add the
        # cell to the dividing cell list to call CompuCell3D's
        # division method later
        dividing_cells.append(cell)
# looping over dividing cells
for cell in dividing_cells:
    # calling CC3D's division method on relevant cells
    self.divide_cell_random_orientation(cell)
```

Listing 27: Example implementation of continuous PhenoCellPy tasks in CompuCell3D. Each CC3D cell has its own cell dictionary (`cell.dict`) that can have custom data. We have saved each cell's Phenotype to the key "phenotype" in the dictionary, see Listing 31.

PhenoCellPy is intended to be used with other python-based modeling frameworks as an embedded model. A PhenoCellPy phenotype should be attached to each relevant agent in the main model, then for each main model agent that has a Phenotype the Phenotype time-step method should be called at every main model time-step. The Phenotype time-step will return boolean flags for cell division, removal from the simulation (*e.g.*, cell death, migration), and cell division, the user is then responsible for performing tasks based on those flags. For instance, in the Necrosis Standard CompuCell3D example (see online `https://github.com/JulianoGianlupi/PhenoCellPy/tree/main/CC3D_examples/Necrosis`), the CompuCell3D cell changes its cell type (see [10] for a definition of cell type in CompuCell3D) when changing from the "hydropic/osmotic swell"[2] Phase to the "lysed"[3] Phase (see Section 5.3.7 for information about the pre-built necrosis Phenotype). Listing 27 shows a generic implementation of continuous PhenoCellPy tasks in a CompuCell3D simulation.

We currently have developed and tested PhenoCellPy embedded models in CompuCell3D [10] (CC3D) and Tissue Forge [82] (TF).

5.2 PhenoCellPy Overview

PhenoCellPy makes the construction of new behavior models easy by breaking them into component parts. The Phenotype class (Section 5.2.3) is the main "container" of behavior. It can contain all the stages of infection, all the stages of the cell cycle, all the phases of necrosis, *etc*. The Phenotype is broken down into component phases, represented by the Phase class (Section 5.2.2). The Phase class contains more specific behaviors, *e.g.*, osmotic swelling (necrosis), cell rupture, volume decrease during apoptosis. The Phase class also defines what should be the volume change rates, and what is the trigger for Phase change.

[2] see BioPortal's ontology definition for hydropic `https://bioportal.bioontology.org/ontologies/PATO?p=classes&conceptid=http%3A%2F%2Fpurl.obolibrary.org%2Fobo%2FPATO_0002119`

[3] see BioPortal's ontology definition for lysed `https://bioportal.bioontology.org/ontologies/PATO?p=classes&conceptid=http%3A%2F%2Fpurl.obolibrary.org%2Fobo%2FPATO_0065001`

Finally the cell volume dynamics are handled by the Cell Volume class (Section 5.2.1). Figure 5.2 shows schematics of how PhenoCellPy is organized.

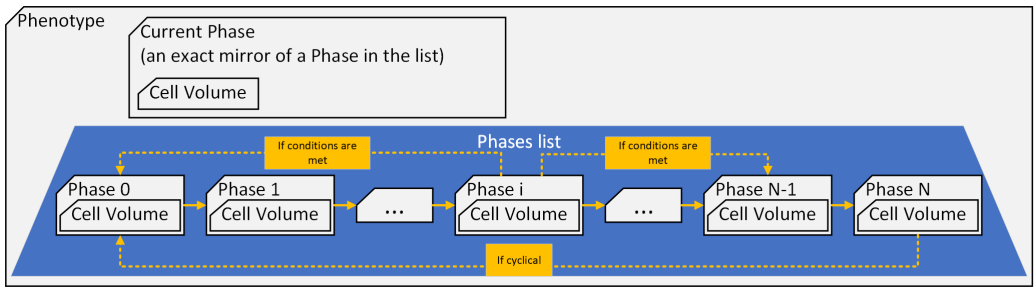


Figure 5.2: How PhenoCellPy is organized.Gray boxes are PhenoCellPy classes, the blue trapezoid is the lists of constituent Phases. Ownership of objects is conveyed through overlaying shapes (*e.g.*, the Phenotype owns the list of Phases, the Phase owns the Cell Volume). Yellow arrows indicate sequence in the list of Phases. Any Phase could go to any other Phase, as dictated by the modeler. A Phase can have exits to any number of Phases, and a Phase can have entrances from multiple Phases.

We will now briefly describe how to use each class and give an overview of how they work. Starting with the Cell Volume class and working our way up to the Phenotype class. In Section 5.2.4, we give a brief example of PhenoCellPy's use. For further detail on how PhenoCellPy is implemented in Python see Supplemental Materials C.1.

5.2.1 Cell Volume Class

The Cell Volume class defines how big the simulated cell is and how much of its volume is taken by the nucleus and cytoplasm. It also separates the cellular volume into fluid and solid fractions, and has a concept of a calcified volume fraction. All the volumes and fractional volumes we define, as well as volume change rates are in Table 5.1. The user of PhenoCellPy should decide if the different volumes should be explicitly included in their model, or if they will use only the cytoplasm and nuclear volumes (without a distinction of solid and fluid fraction), or, simply, the total cell volume. The default volumes used by PhenoCellPy are from MCF-7 cells [84] in cubic microns.

The volume dynamics model works by relaxing the dynamic volumes (marked with * in Table 5.1) to their target volumes using their respective volume change rates with a system

of ODEs (Equations 5.1). Then the other volumes are set as relations of the dynamic volumes. It is important to note that, while the volumes and target volumes are attributes of the Cell Volume class, the volume change rates are not, they are an attribute of the Phase class. We made this separation because how fast a cell changes its volume is a property of its state (Phase in PhenoCellPy), therefore the change rates are an attribute of the Phase class. Equations 5.1 show ODE system governing the dynamic volumes, in those equations the superscript tg denotes target. Equations 5.2 show how the other volumes are calculated from the dynamic volumes and each other.

$$\frac{dV_F}{dt} = r_F \ (f_F^{tg} \times V - V_F) \ , \tag{5.1a}$$

$$\frac{dV_{NS}}{dt} = r_{NS}(V_{NS}^{tg} - V_{NS}) \ , \tag{5.1b}$$

$$\frac{dV_{CS}}{dt} = r_{CS}(f_{CN}^{tg} \times V_{NS} - V_{CS}) \ , \tag{5.1c}$$

$$\frac{df_C}{dt} = r_C \ (1 - f_C) \ . \tag{5.1d}$$

$$V_{NF} = f_F \times V_N \ , \tag{5.2a}$$

$$V_{CF} = V_F - V_{NF} \ , \tag{5.2b}$$

$$V_S = V_{NS} + V_{CS} \ , \tag{5.2c}$$

$$V_N = V_{NS} + V_{NF} \ , \tag{5.2d}$$

$$V_C = V_{CS} + V_{CF} \ , \tag{5.2e}$$

$$V = V_N + V_C \ , \tag{5.2f}$$

$$f_F = V_F/V \ , \tag{5.2g}$$

$$f_{CN} = V_C/V_N \ . \tag{5.2h}$$

Note that Equation 5.1d implies that the target calcified fraction is always 1, we are adopting this default behavior from PhysiCell [11]. For most Phases predefined in PhenoCellPy the calcification rate is zero.

Volume type	Symbol	Change Rate	Symbol
Total	V	Fluid	r_F
Total Nuclear	V_N	Nuclear Solid	r_{NS}
Nuclear Solid	V_{NS}^*	Cytoplasm Solid	r_{CS}
Nuclear Fluid	V_{NF}	Calcified Fraction	r_C
Total Cytoplasm	V_C		
Cytoplasm Solid	V_{CS}^*		
Cytoplasm Fluid	V_{CF}		
Cytoplasm to nuclear ratio	f_{CN}		
Total Fluid	V_F^*		
Fluid Fraction	f_F		
Calcified Fraction	f_C^*		

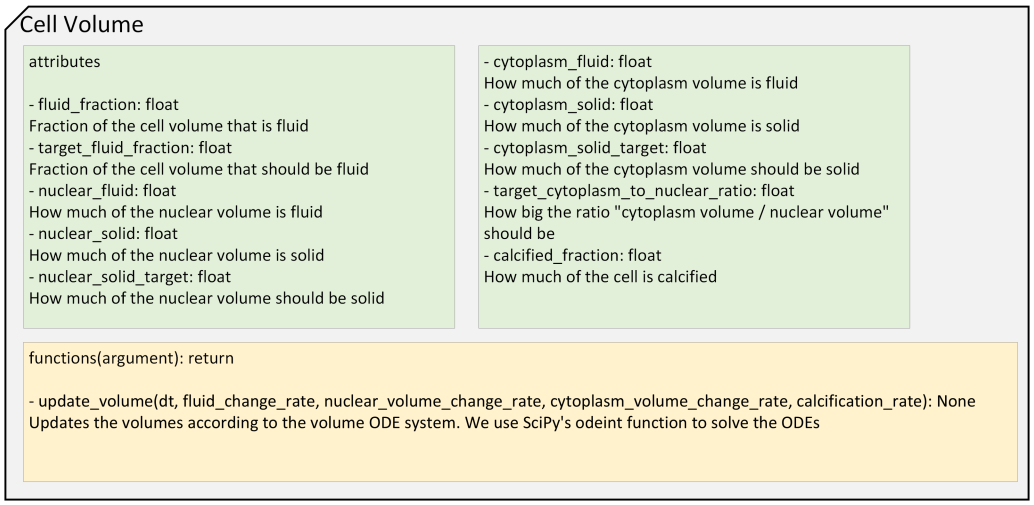
Table 5.1: Volumes and volume change rates defined by the Cell Volume class. Volumes marked with * are the dynamic volumes.

Figure 5.3 shows the Cell Volume class attributes and functions. It also schematizes how the volume update function works. It uses the volumes attributes from the Cell Volume class together with the volume change rates passed to it by the Phase class to update the volumes of the Cell Volume class.

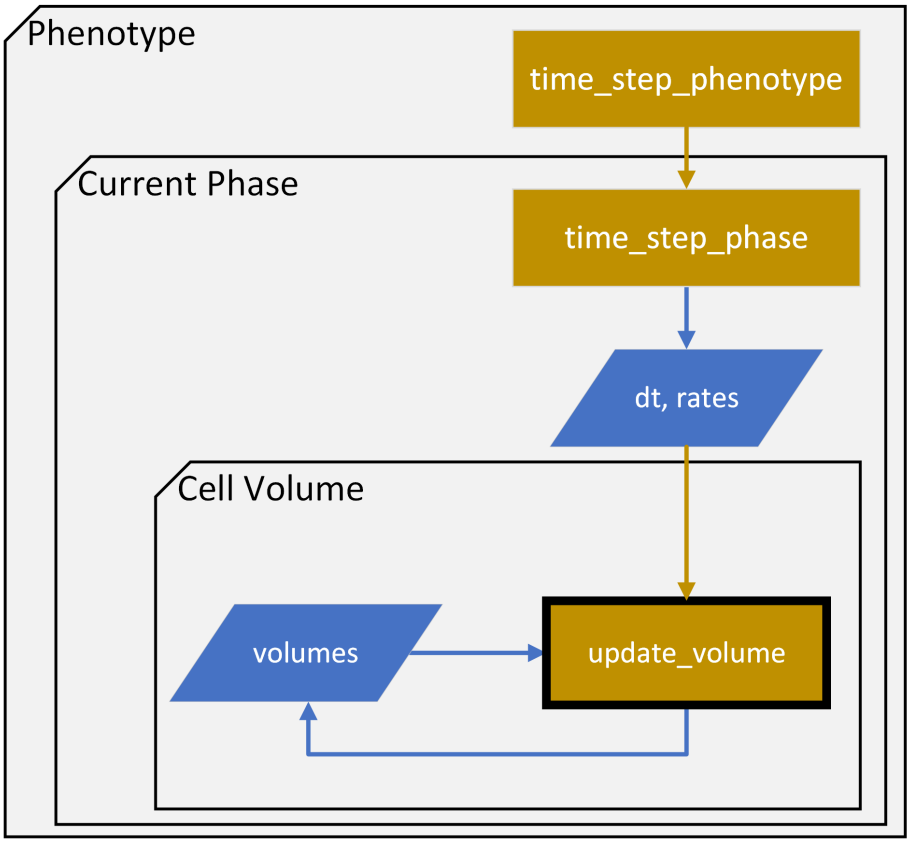
5.2.2 Phase Class

The phase is the ”base unit” of the phenotype. It defines volume change rates, checks for Phase change, and performs phase-specific tasks on phase entry, phase exit and during each time-step.

The phase change check and phase specific tasks are user-definable functions. For flexibility, we impose that all user-definable functions **must** be able to take any number of arguments as inputs (*i.e.*, be a Python *args function). Otherwise, the way PhenoCellPy calls those functions would have to change as the number of arguments changes. The user



(a) Cell Volume class attributes and functions



(b) Cell Volume time-step

Figure 5.3: Cell Volume class attributes and functions (5.3a), and Cell Volume update volume (5.3b). The Cell Volume update is highlighted by the black outline. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, green diamonds are decisions. Yellow arrows mean function call, blue arrows are information being passed.

can define their function is such a way that the *args are unused or unnecessary. Our default transition functions and entry/exit functions functions do not use the args. For PhenoCellPy's *alpha* version the modeler is responsible for defining, transferring and updating the arguments, creating interface APIs to facilitate this will be included with PhenoCellPy by release.

Examples of phase specific tasks are doubling the target volumes when entering the proliferating Phase (see Sections 5.3.2 and 5.3.3), or halving the target volumes after division. In-Phase tasks could be checking if the quantity of a nutrient a cell has access to. Figure 5.4a shows the Phase class attributes and functions, and Figure 5.4b the flowchart for the Phase class time-step.

5.2.2.1 Phase transition

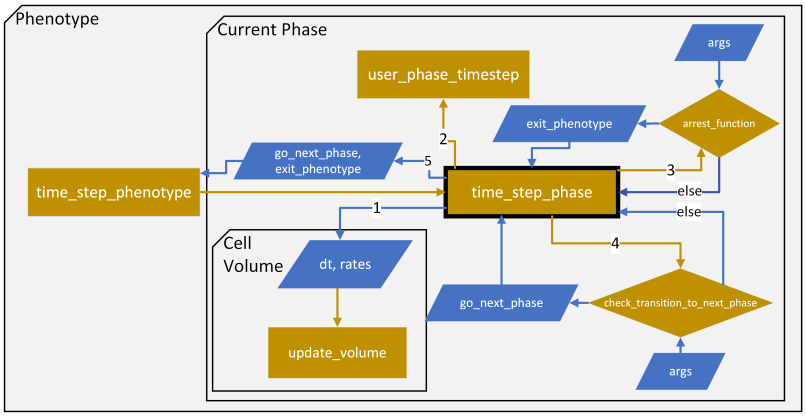
PhenoCellPy has two pre-defined transition functions that are used by our pre-built models, the deterministic phase transition and the stochastic phase transition. For the deterministic transition the Phase evaluates if the time spent in this Phase (T) is greater than the Phase period (τ), Equation 5.3. For the stochastic case we draw a probability from a Poisson distribution for a single event occurring (Equation 5.4). The Poisson probability depends on the time-step length (dt) and expected Phase period (τ). We cannot approximate $1 - e^{-dt/\tau} \approx dt/\tau$ in Equation 5.4, as we don't know what values of dt and τ the modeler will use.

$$P(A \to B) = \begin{cases} 1 & \text{if } T > \tau \\ 0 & \text{else} \end{cases} . \tag{5.3}$$

$$P(A \to B) = 1 - e^{-dt/\tau} . \tag{5.4}$$

The user can define their own transition functions that may depend on any number of simulation parameters, *e.g.*, oxygen levels, having a neighboring cell, signaling molecules.

(a) Phase class attributes and functions



(b) Phase time-step

Figure 5.4: Phase class attributes and functions (5.4a), and Phase time-step flowchart (5.4b). The time-step function is highlighted by the black outline, the order in which it performs operations is overlayed on the arrows. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, yellow diamonds are decision-making functions. Yellow arrows mean function call, blue arrows are information being passed.

One of our CompuCell3D [10] examples uses a custom transition function, Ki-67 Basic Cycle Improved Division (Section 5.4.1.1), to ensure cells don't divide early. The custom transition function checks that the in-simulation volume of the simulated cell has reached the doubling volume, Equation 5.6.

If the Phase can transition to several Phases (instead of just one), the user can use the transition function to select which Phase to go to.

5.2.3 Phenotype Class

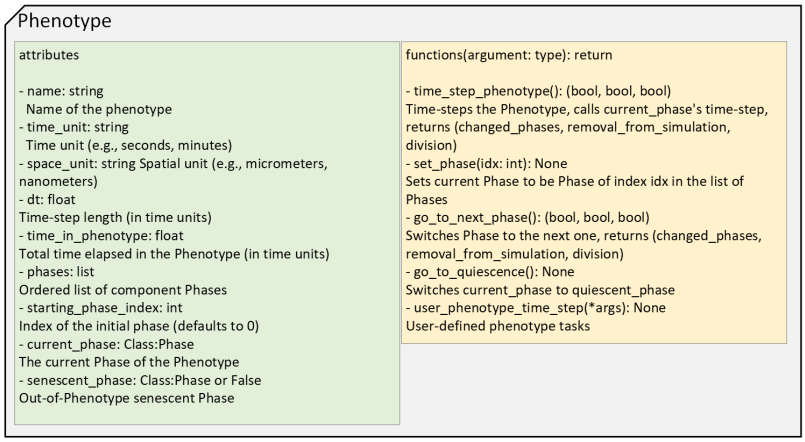
The Phenotype is the main concept of PhenoCellPy, in PhenoCellPy "Phenotype" means any sequence of distinct cell behaviors. For instance, a quiescent-proliferating cell cycle is a phenotype with two phases (quiescence, and growth/division); the necrotic phenotype starts with a osmotic swelling phase, followed by dissolution of the cell after it bursts.

PhenoCellPy supports cyclical and acyclical Phenotypes, as well as Phenotypes with an arbitrary sequence of Phases. To facilitate a Phenotype end-point we have defined a method to exit the Phenotype cycle and go into a special senescent Phase. To make use of this functionality, the modeler has to define the optional arrest function which is a member of the Phase class (Section 5.2.2). The arrest function is called by the Phase time-step function (Appendix C.1.1.2) and its return value is used by the Phenotype time-step function to exit the Phenotype cycle.

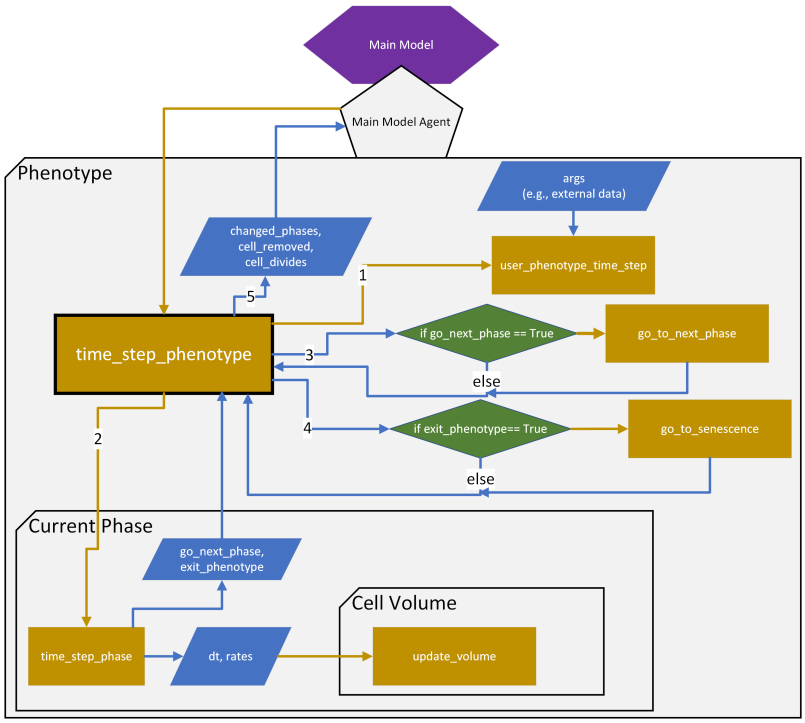
The Phenotype class owns the list of all Phases that make it. It switches Phases or goes to the senescent Phase when it receives the respective signals from the Phase time-step, and performs specific user-defined Phenotype tasks each time-step. Figure 5.5a shows the Phenotype class attributes and functions, and Figure 5.5b the flowchart for its time-step.

5.2.4 Using PhenoCellPy

PhenoCellPy's intended use is as an embedded model, meaning it should be loaded into some other modeling platform, *e.g.*, CompuCell3D [10], Tissue Forge [82]. CompuCell3D



(a) Phenotype class attributes (green box) and functions (yellow box).



(b) Phenotype time-step

Figure 5.5: Phenotype class attributes and functions (5.5a), and Phenotype time-step flowchart (5.5b). The time-step function is highlighted by the black outline, the order in which it performs operations is overlayed on the arrows. Purple hexagon represents the model PhenoCellPy is embedded in (*i.e.*, the main model), the gray pentagon is an agent from the main model. Gray boxes are PhenoCellPy classes, yellow rectangles are functions being called, blue parallelograms are information being passed to/from functions, green diamonds are decisions. Yellow arrows mean function call, blue arrows are information being passed.

already supports other modeling frameworks as embedded models, namely SBML [12], Antimony [85], and MaBoSS [86].

To use PhenoCellPy, the modeler has to import PhenoCellPy and should initialize a Phenotype object at the simulation start. The Phenotype initialization (in the version of the package at time of writing) takes as arguments:

- The phenotype name
- Time-step length (dt)
- Time and space units
- The list of Phases that make the Phenotype
- Lists for the Phases' target volumes, initial volumes, volume change rates
- The initial volume of the simulated cell
- The starting Phase index
- The senescent Phase
- User-defined Phenotype time-step tasks and initial arguments for them
- List of user-defined phase time-step tasks and initial arguments for them

All arguments, except the time-step period, for Phenotype initialization are optional. Listing 28 show the import and initialization of a pre-built Phenotype.

```
# importing PhenoCellPy
import PhenoCellPy as pcp
# defining the time-step period for pcp
dt = 1
# initialing a phenotype and saving it to a handle
ki67_basic = pcp.phenotypes.Ki67Basic(dt=dt)
```

Listing 28: Initialization of a pre-built Phenotype using the default argument values explicitly.

5.2.4.1 *Initialization*

```
import PhenoCellPy as pcp
dt = .1  # min, defining the time-step
# defining the 1st Phase and setting its parameters
stable_phase_0 = pcp.phases.Phase(index=0,
                    previous_phase_index=-1, next_phase_index=1,
                    dt=dt, name="stable0", division_at_phase_exit=False,
                    removal_at_phase_exit=False, fixed_duration=True,
                    phase_duration=10)
# defining the custom transition function for the 2nd phase
def grow_phase_transition(*args):
    # grabbing arguments for clarity
    volume = args[0]
    doubling_volume = 0.8 * args[1] # we use a bit less than the
    # doubling volume to guarantee a transition
    time_phase = args[2]
    phase_duration = args[3]
    # checking for transition
    return volume >= doubling_volume and time_phase > phase_duration
# 2nd Phase, initial args that will fail the check as a safe-guard
grow_phase = pcp.phases.Ki67Positive(index=1,
                previous_phase_index=0, next_phase_index=2,
                dt=dt, name="grow", fixed_duration=True,
                phase_duration=50, entry_function=None,
                entry_function_args=[None],
                check_transition_to_next_phase_function=\
                    grow_phase_transition,
                check_transition_to_next_phase_function_args=\
                    [0, 9, 0, 9])
# defining the 3rd Phase
stable_phase_1 = pcp.phases.Phase(index=2,
                    previous_phase_index=1, next_phase_index=3,
                    dt=dt, name="stable1", fixed_duration=True,
                    phase_duration=5)
```

Listing 29: Initialization of a custom Phenotype. Part 1.

```
# defining the custom transition function for the 4th phase
def shrink_phase_transition(*args):
# saving the arguments to variables for clarity
    dt = args[0]
    phase_duration = args[1]
    total = args[2]
    total_target = args[3]
    # doing the transition checks. Stochastic and volume repectively
    time_check = np.random.uniform() < \
        (1 - np.exp(-dt / phase_duration))
    volume_check = total <= 1.1 * total_target
    return time_check and volume_check
# 4th phase, we use initial args that will fail the check as a safe-guard
shrink_phase = pcp.phases.Ki67PositivePostMitotic(index=3,
                previous_phase_index=2, next_phase_index=0,
                dt=dt, name="shrink", phase_duration=100,
                entry_function=None, entry_function_args=[None],
                check_transition_to_next_phase_function= \
                    shrink_phase_transition,
                check_transition_to_next_phase_function_args= \
                    [0, 1, 99, 0])
# defining the phenotype
custom_phenotype = pcp.Phenotype(name="oscillate volume with rests",
                        dt=dt,
                        time_unit="min", space_unit="micrometer",
                        phases= \
                            [custom_p0, custom_p1,
                            stable_phase_1, shrink_phase],
                    senescent_phase=False, starting_phase_index=0,
                    user_phenotype_time_step=None,
                    user_phenotype_time_step_args=[None, ])
```

Listing 30: Initialization of a custom Phenotype. Part 2.

A user can also define their own Phenotype, to do so they have to initialize each constituent Phase of the Phenotype and pass them as a list to the Phenotype object initialization. Listing 29 and 30, shows an example of a custom Phenotype being built, some of the Phases use custom transition functions. The example in Listing 29 and 30 only passes the necessary attributes to the classes.

After having initialized the Phenotype the user has to attach it to each agent that will

use that Phenotype. How to do this attachment is modeling platform dependent, in CompuCell3D the recommended method is to make the Phenotype a cell dictionary (Listing 31), we have created an utility function that does this (Listing 32). In Tissue Forge recommended method is to have a dictionary with cell ids as keys and Phenotype as items (Listing 33).

```
# looping over CC3D cells
for cell in self.cell_list:
    # adding a phenotype to the cell dictionary
    cell.dict["phenotype"] = ki67_basic.copy()
```

Listing 31: Attaching a Phenotype to a CompuCell3D cell. Listing 28 shows the phenotype initialization.

```
# looping over CC3D cells
for cell in self.cell_list:
    # adding a phenotype to the cell using the utility function
    pcp.utils.add_phenotype_to_CC3D_cell(cell, ki67_basic)
```

Listing 32: Attaching a Phenotype to a CompuCell3D cell using PhenoCellPy's utility function. Listing 28 shows the phenotype initialization.

```
# creating an empty dictionary to keep the phenotypes in
cells_phenotypes = {}
# looping over TF agents of type Cell
for cell in Cell.items():
    # adding the phenotype to the dictionary using the cell id as key
    cells_phenotypes[f"{cell.id}"] = ki67_basic.copy()
```

Listing 33: Attaching a Phenotype to a Tissue Forge cell. Listing 28 shows the phenotype initialization.

5.3 Pre-defined Phenotypes

PhenoCellPy comes with several pre-packadged Phenotypes defined. As with the Methods Section (Section 5.2), we've removed most docstrings and comments from code presented in this Section.

5.3.1 Simple Live Cycle

Simplest Phenotype defined, it is a cell-cycle Phenotype of one single Phase. The transition from one Phase to the next (which is the same Phase) is stochastic with an expected duration of $\approx 23h$, the expected cycle time for a MCF-10A cell line cell [87].

5.3.2 Ki-67 Basic

The Ki-67 Basic is a two Phase cell-cycle phenotype, it matches experimental data that uses Ki-67, a protein marker for cell proliferation [88]. A cell with a positive Ki-67 marker is in its proliferating state, if there's no Ki-67 it is in quiescence.

The quiescent Phase (Ki-67 negative) uses an expected Phase duration of $4.59h$, transition from this Phase is set to be stochastic. The proliferating Phase (Ki-67 positive) uses a fixed duration (*i.e.*, the transition is deterministic) of $15.5h$. We utilize the same reference values for Phase durations as PhysiCell [11].

5.3.3 Ki-67 Advanced

Ki-67 Advanced adds a post-mitosis Phase to represent the time it takes Ki-67 to degrade post cell division. The proliferating Phase and Ki-67 degradation Phase are both deterministic, with durations of $13h$ and $2.5h$, respectively. The quiescent (Ki-67 negative) Phase uses the stocastic transition, with an expected duration of $3.62h$. Again, we utilize the same reference values for Phase durations as PhysiCell [11].

5.3.4 Flow Cytometry Basic

Flow Cytometry Basic is a three Phase live cell cycle. It represents the $G0/G1 \rightarrow S \rightarrow G2/M \rightarrow G0/G1$ cycle. The $G0/G1$ phase is more representative of the quiescent phase than the first growth phase, as no growth occurs in this Phase. The Phenotype transitions stochastically from this phase, its expected duration is $5.15h$. The S phase is the phase responsible for doubling the cell volume, transition to the next phase is stochastic, its expected duration is $8h$. The cell volume growth rate is set to be [total volume growth]/[phase duration]. The G2/M is the pre-mitotic rest phase, the cell divides when exiting this phase. Transition to the next phase is stochastic, its expected duration is 5h. Reference phases durations are from "The Cell: A Molecular Approach. 2nd edition" [89].

5.3.5 Flow Cytometry Advanced

Flow Cytometry Advanced is a four Phase live cell cycle. It represents the $G0/G1 \rightarrow S \rightarrow G2 \rightarrow M \rightarrow G0/G1$ cycle. The behaviors The mechanics of the Phases are the same as in Flow Cytometry Basic (Section 5.3.4) with an added rest Phase (the separation of $G2$ from M). All Phase transitions are stochastic, and their expected durations are: $4.98h$, $8h$, $4h$, and $1h$, respectively. Cell division occurs when exiting Phase M.

5.3.6 Apoptosis Standard

Apoptosis Standard is a single Phase dead Phenotype. The cell in this Phenotype sets $V_{NS}^{tg} = 0$, $f_{CN}^{tg} = 0$, and $f_F = 0$ on Phase entry. By doing this, the cell will diminish in volume until it disappears, the rate used for the cytoplasm reduction is $r_{CS} = 1/60\ \mu m^3/min$, for the nucleus it is $r_{NS} = 0.35/60\ \mu m^3/min$, and the fluid change rate is $r_F = 3/60\ \mu m^3/min$, volume change rates reference values from [90, 91]. This Phenotype Phase uses a fixed duration of $8.6h$, the simulated cell should be removed from the simulation domain when the Phase ends.

5.3.7 Necrosis Standard

Necrosis Standard is a two Phase dead Phenotype. The first phase represents the osmotic swell of a necrotic cell, it doesn't have a set or expected phase duration, instead it uses a custom transition function that monitors the cell volume and transition to the next phase happens when the cell total volume (V_T) reaches its rupture volume (V_R), set to be twice the original volume by default, see Equation 5.5 and Listing 34. On Phase entry, the solid volumes targets are set to zero (*i.e.*, $V_{CS}^{tg} = V_{NS}^{tg} = 0\mu m$), the target fluid fraction is set to one ($f_F^{tg} = 1$), and the target cytoplasm to nuclear ratio is set to zero ($f_{CN}^{tg} = 0$). These changes cause the cell to swell. The rates of volume change are: solid cytoplasm $r_{CS} = 3.2/60 \times 10^{-3}\ \mu m^3/min$, solid nucleus $r_{NS} = 1.3/60 \times 10^{-2}\ \mu m^3/min$, fluid fraction $r_F = 6.7/60 \times 10^{-1}\ \mu m^3/min$, calcified fraction $r_C = 4.2/60 \times 10^{-3}\ \mu m^3/min$.

The second Phase represents the ruptured cell, it is up to the modeler and modeling framework how the ruptured cell should be represented. For instance, in Tissue Forge [82] the ruptured cell should become several fragment agents, whereas in CompuCell3D [10] the fragmentation can be achieved by setting the ruptured cell contact energy with the medium to be negative. The cell fragments shrink and dissolve into the medium. As a safeguard, this Phase uses a deterministic transition time of 60 days, after which the fragments are flagged for removal. On Phase entry all target volumes are set to zero. The volume change rates are: solid cytoplasm $r_{CS} = 3.2/60 \times 10^{-3}\ \mu m^3/min$, solid nucleus $r_{NS} = 1.3/60 \times 10^{-2}\ \mu m^3/min$, fluid fraction $r_F = 5/60 \times 10^{-1}\ \mu m^3/min$, calcified fraction $r_C = 4.2/60 \times 10^{-3}\ \mu m^3/min$.

$$P(\text{Swelling} \rightarrow \text{Ruptured}) = \begin{cases} 1 & \text{if } V_T > V_R \\ 0 & \text{else} \end{cases}. \tag{5.5}$$

```
def _necrosis_transition_function(self, *none):
    """
    Custom phase transition function. The simulated cell should only
    change phase once it bursts (i.e., when its volume is above the
    rupture volume), it cares not how long or how little time it
    takes to reach that state.

    :param none: Not used. This is a custom transition function,
    therefore it has to have args, i.e., the function implements
    the PhenoCellPy interface

    :return: Flag for phase transition
    :rtype: bool
    """
    return self.volume.total > self.volume.rupture_volume
```

Listing 34: Necrotic Standard's osmotic swelling Phase transition function.

5.4 Selected Examples

5.4.1 CompuCell3D

For CompuCell3D [10] the reccomended method of using PhenoCellPy is to initialize the Phenotype in the steppable start function, and add it as a cell dictionary entry (see Listing 31). Then, in the steppable step function, loop over cells that have a Phenotype model, call the Phenotype time-step, and use the time-step return values as needed (see Listing 27).

See CompuCell3D's manual [79] for more information on steppables, cell dictionaries, and other CompuCell3D concepts.

5.4.1.1 Ki-67 Basic Cycle Improved Division

The stock Ki-67 Basic Cycle causes CompuCell3D cells to behave in non-biological ways. This happens because Cellular Potts Model [7] (CompuCell3D's paradigm) cells are made of several voxels [10], and the proliferating Phase of Ki-67 has a set period. This means that the simulated cells in CompuCell3D may not grow to the doubling volume before they

are flagged for cell division. This means that the median cell volume of the cells in the simulation decreases.

Ki-67 Basic Cycle Improved Division fixes this by defining a custom transition function, see Listing 35. It still is a deterministic transition function, however, in addition of monitoring T (time spend in phase), it also monitors the simulated cell volume (V_S) and checks if it is bigger than the doubling volume (V_D). The custom transition function evaluates the transition probability as,

$$P(\text{division}) = \begin{cases} 1 & \text{if } T > \tau \wedge V_S \geq V_D \\ 0 & \text{else} \end{cases} . \tag{5.6}$$

The full implementation of this example is in Supplemental Materials C.4.

```
def Ki67pos_transition(*args):
    # args = [cc3d cell volume, phase's target volume, time in
    phase, phase duration
    return args[0] >= args[1] and args[2] > args[3]
```

Listing 35: Ki-67 Basic Cycle Improved Division transition function

5.4.2 Tissue Forge

For Tissue Forge [82] the suggested method of using PhenoCellPy is to initialize the Phenotype in a similar way to Tissue Forge's cell initialization, keep a dictionary of agent IDs as keys and Phenotype models as values, and create a Tissue Forge event [82] to time-step the Phenotype objects. A generic implementation of PhenoCellPy within Tissue Forge can be found in Listing 36 and 37. Supplemental Information C.5 shows the implementation of the Ki-67 Basic Cycle Phenotype.

```python
import numpy as np
import tissue_forge as tf
import PhenoCellPy as pcp
# dimensions of universe
dim = [10., 10., 10.]
# new simulator
tf.init(dim=dim)
# defining the inter-cell potential
pot = tf.Potential.morse(d=3, a=5, min=-0.8, max=2)
# define Cell class
mass = 40
class CellType(tf.ParticleTypeSpec):
    mass = mass
    target_temperature = 0
    radius = radius
    dynamics = tf.Overdamped
# initialize cell
Cell = CellType.get()
# bind the potential with the *TYPES* of the particles
tf.bind.types(pot, Cell, Cell)
# uniform random cube
positions = np.random.uniform(low=0, high=10, size=(10, 3))
# place cells at the positions
for pos in positions:
    Cell(pos)
# initialize Phenotype
ki67_basic = pcp.phenotypes.Ki67Basic(dt=dt)
# Pair the cells ids with the Phenotype model
global cells_cycles
cells_cycles = {}
for cell in Cell.items():
    cells_cycles[f"{cell.id}"] = ki67_basic.copy()
```

Listing 36: Generic implementation of PhenoCellPy within Tissue Forge. Part 1

```
# Defining the Phenotype stepping event
def step(event):
    for cell in Cell.items():
        pcycle = cells_cycles[f"{cell.id}"]
        pcycle.current_phase.simulated_cell_volume = p.mass * density
        phase_change, removal, division = pcycle.time_step_phenotype()
        if phase_change:
            phase_change_tasks(cell)
        if removal:
            removal_tasks(cell)
        if division:
            division_tasks(cell)
        other_tasks(cell)
    return 0
# running the time-stepping event every time-step
tf.event.on_time(invoke_method=step_cycle_and_divide,
                 period=.9*tf.Universe.dt)
# run the simulator interactive
tf.run()
```

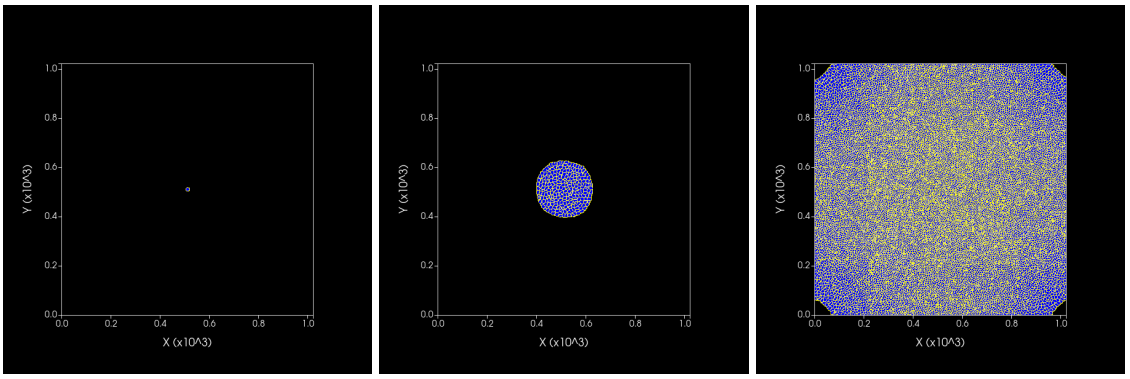
Listing 37: Generic implementation of PhenoCellPy within Tissue Forge. Part 2

5.5 Selected Results

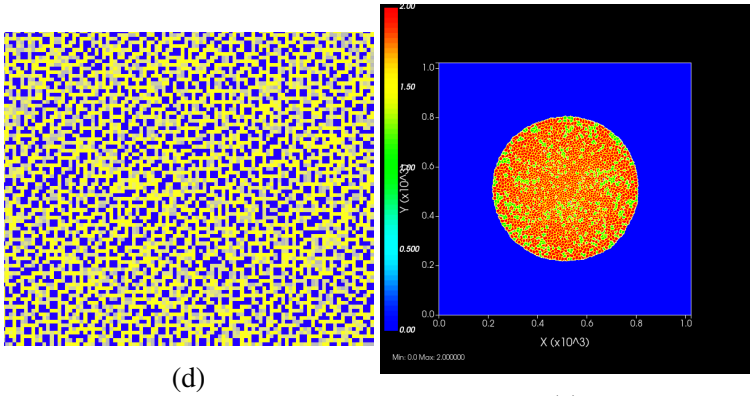
In this section, we will present selected results from some of the pre-built CompuCell3D and Tissue Forge examples.

5.5.1 CompuCell3D

5.5.1.1 Ki-67 Basic Cycle



(a) (b) (c)



(d) (e)

Figure 5.6: Spatial figures for the CompuCell3D simulation using the regular Ki-67 Basic cycle. a, b, and c) shows the cell cluster with no color overlay. They show the cluster at the start of the simulation, step 2000 and step 4000 respectively. d) Zoom in on the center of image c. e) Color coded cells based on the phase they are in, green for the quiescent phase and red for the proliferating phase. Step 2800.

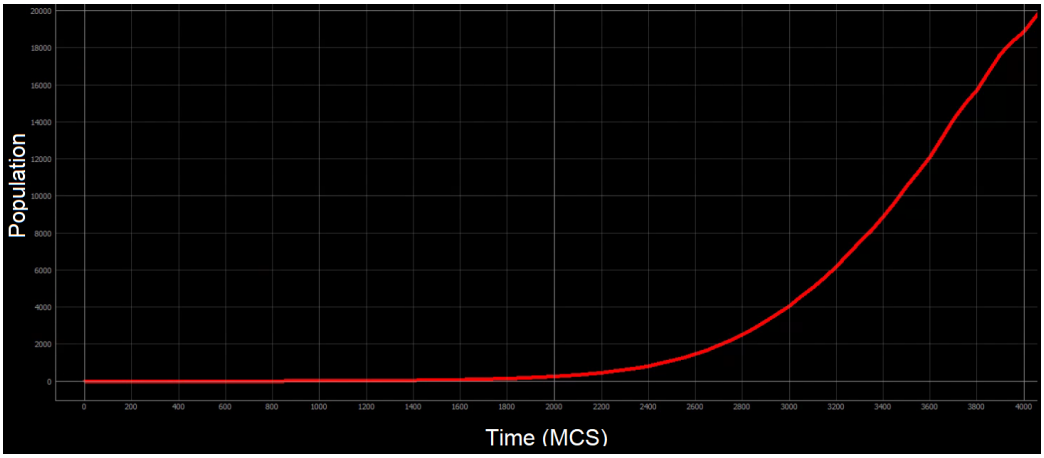
We ran the regular Ki-67 Basic cycle (see Section 5.3.2) and compared it to the Ki-67 Basic cycle improved division (see Section 5.4.1.1) in CompuCell3D. The simulation here is 2D. In this simulation we start with single cell with a volume of 100 pixels (the

pixel to μm conversion is set to $24.94 \mu m/pixel$), therefore they should reach 200 pixels before division. We are using a time-step of $5min/step$, the usual name for the time-step in CompuCell3D, for historical reasons, is Monte-Carlo Step (MCS). With the regular cycle We see that the median and minimum cell volumes go down in time (Figure 5.7b), that happens because the mitotic phase transition happens after a set amount of time no matter the cell volume. As cells grow slowly in CompuCell3D they end up being halved before reaching their doubling volume. We see that this effect is worse towards the center of the cell cluster (see Figure 5.6d), as those cells don't have room to grow at all. In Figure 5.6d we can see that many cells are only a few pixels big. We also see that the spatial phase distribution is random in space (see Figure 5.6e).

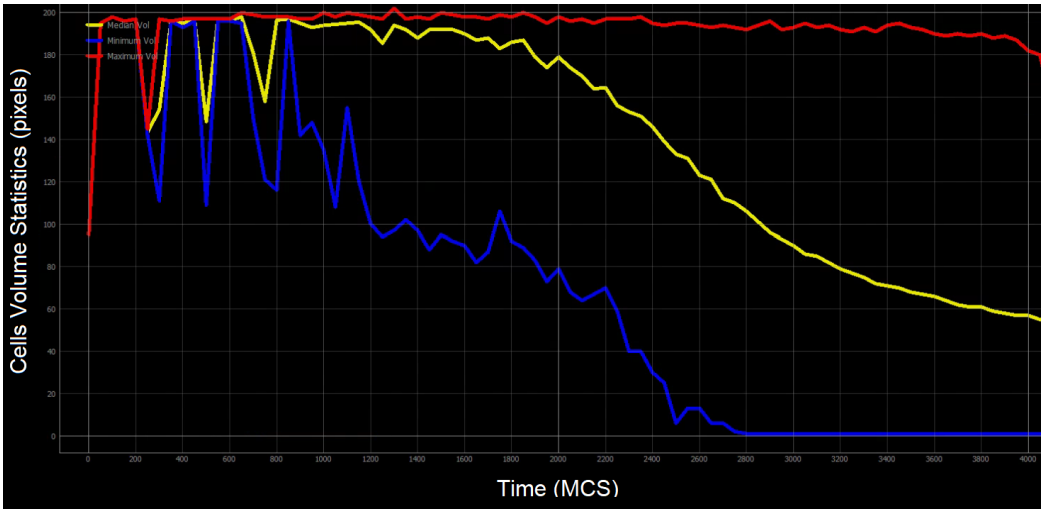
In contrast, when using the modified transition we account for the simulated cell volume. Now the reduction in the median volume is small and happens due to overcrowding and cells pushing each other, and that the total population growth was much smaller (3800 cells at step 4000, see Figure 5.9a, versus 19000, see Figure 5.7a). We do not see overly small cells (see Figure 5.6d). We also see that cells towards the center of the cell cluster stay in the proliferating phase (see Figure 5.8e), unable to reach their doubling volume and divide.

5.5.1.2 Necrosis standard

We ran the necrotic phenotype in a 2D CompuCell3D simulation, we simulate 170 cells plated in a petri dish. We select ten of those cells to undergo necrosis. We keep the initial volume of the cells 100 pixels (with the same pixel to μm relation) and the same time-step duration of $5min/step$. The necrotic cells increase in volume linearly (see Figure 5.11) during the hydropic (osmotic) swell phase until they reach their bursting volume. After bursting (see Figure 5.10) their volume decreases rapidly at first, going near zero almost immediately, and then keeps decreasing more slowly (see Figure 5.11). This happens because the PhenoCellPy model sets the target volume of the cell to 0 when entering the ruptured

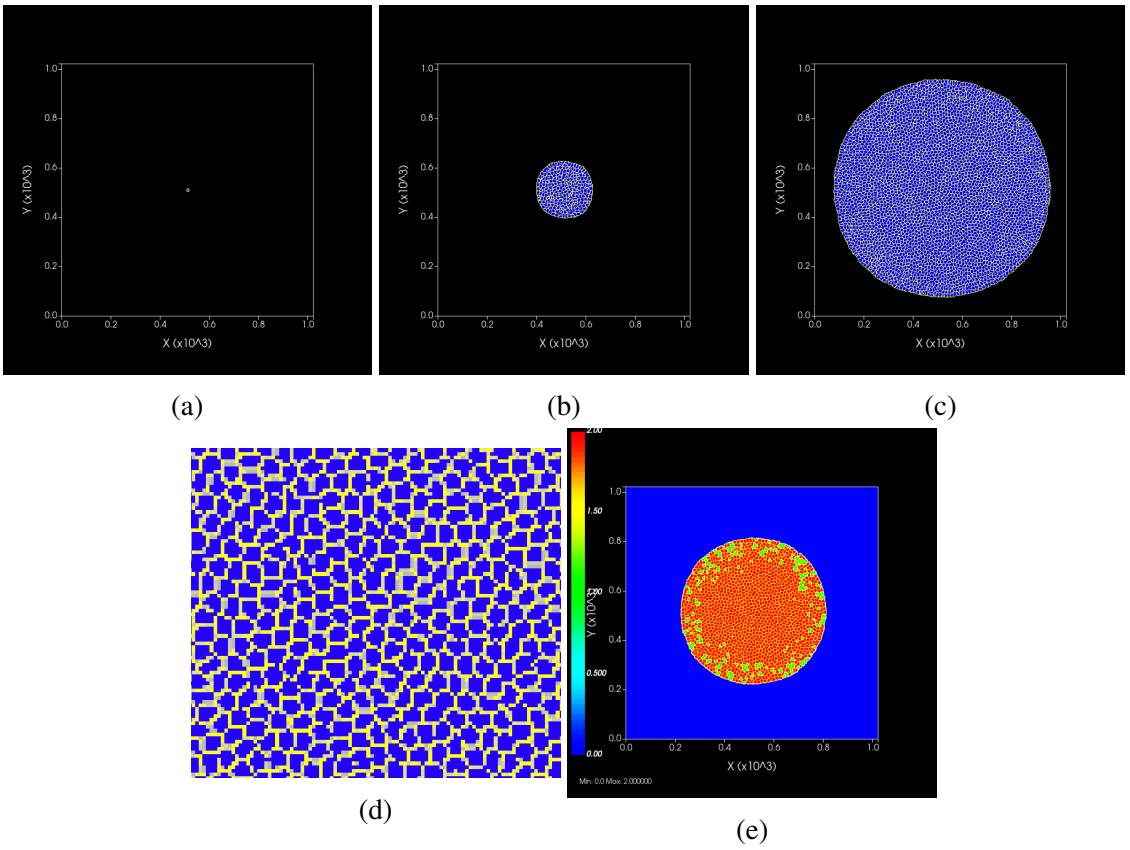


(a)



(b)

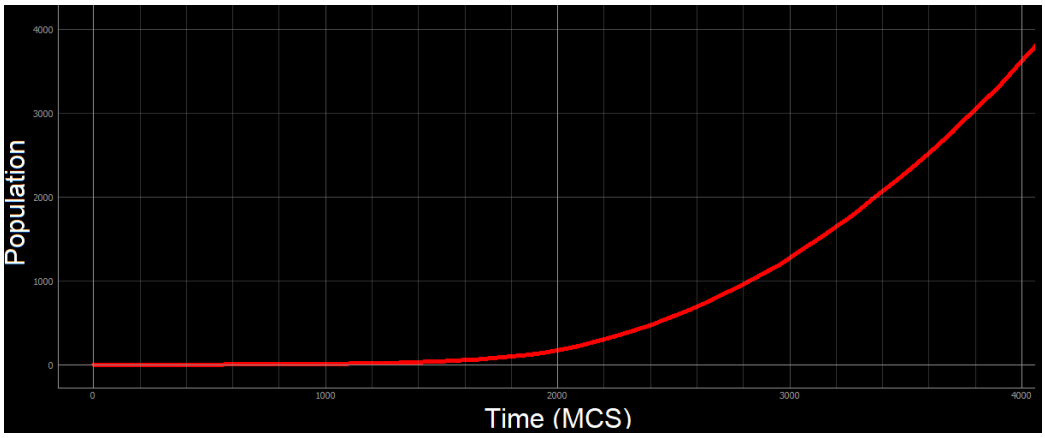
Figure 5.7: Statistics for the cell population in CompuCell3D using the standard Ki-67 Basic cycle. a) Total cell population. b) Cell population volume statistics. Maximum cell volume in red, median in yellow, and minimum in blue.



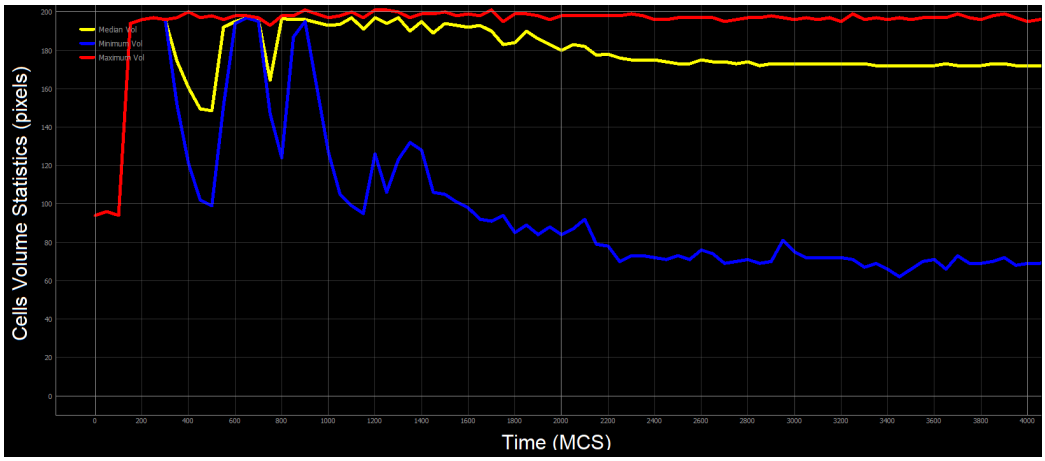
(a) (b) (c)

(d) (e)

Figure 5.8: Spatial figures for the CompuCell3D simulation using the regular Ki-67 Basic cycle. a, b, and c) shows the cell cluster with no color overlay. They show the cluster at the start of the simulation, step 2000 and step 4000 respectively. d) Zoom in on the center of image c. e) Color coded cells based on the phase they are in, green for the quiescent phase and red for the proliferating phase. Step 3200.



(a)



(b)

Figure 5.9: Statistics for the cell population in CompuCell3D using the modified Ki-67 Basic cycle. a) Total cell population. b) Cell population volume statistics. Maximum cell volume in red, median in yellow, and minimum in blue.

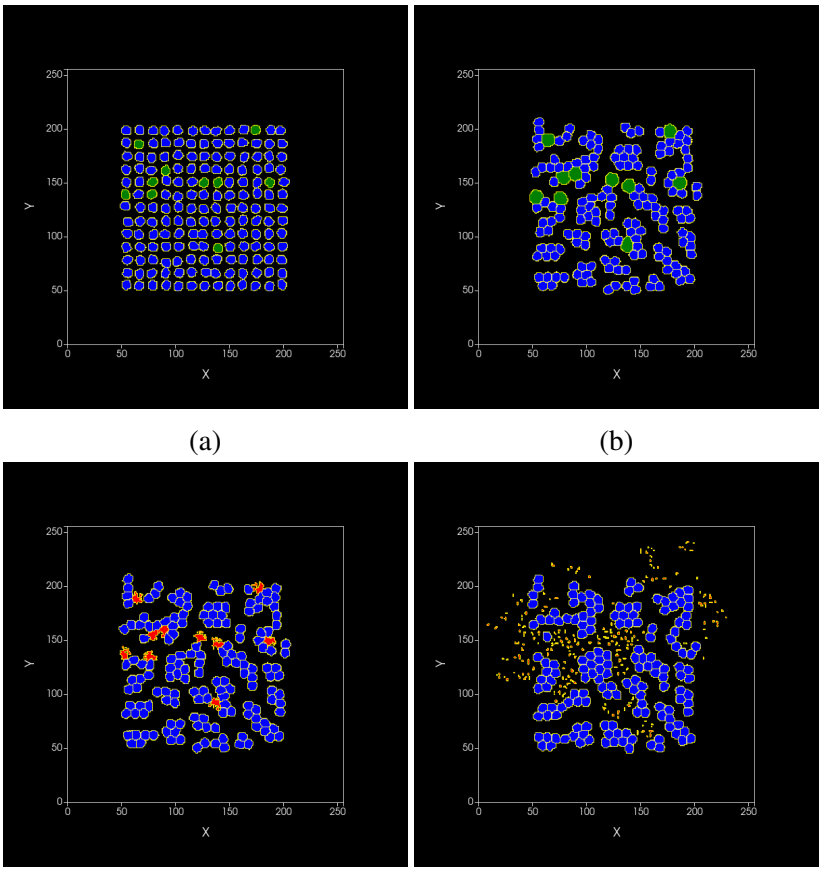


Figure 5.10: Snapshots of the CompuCell3D simulation for the necrotic phenotype, necrotic cells in green, healthy cells in blue, fragmented cells in red. a) simulation start. b) Necrotic cells at maximum volume. c) Moment the necrotic cells bursts. d) end of the simulation.

phase, however, CompuCell3D works by minimizing the energy of the system, and the cell fragments have a negative contact energy with the medium (see CompuCell3D's [10] manuals for a definition of the contact energy).

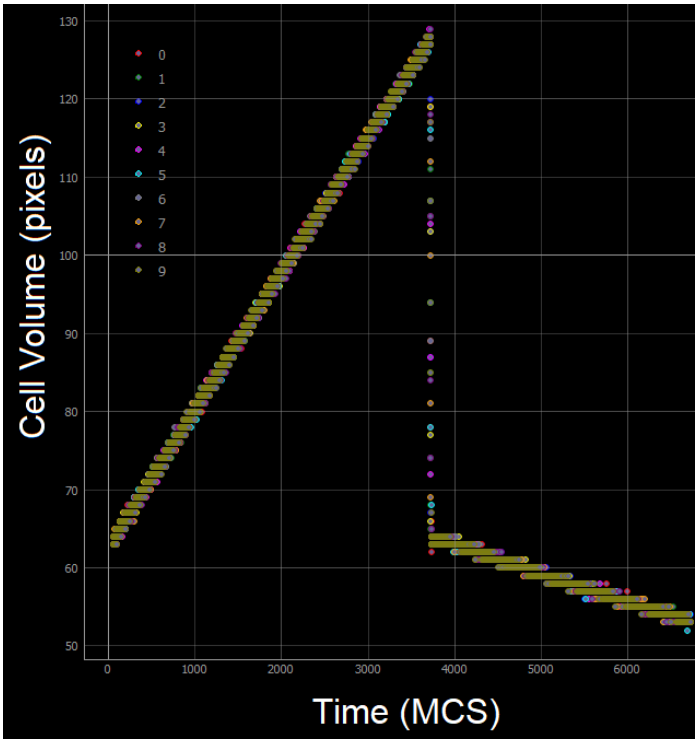


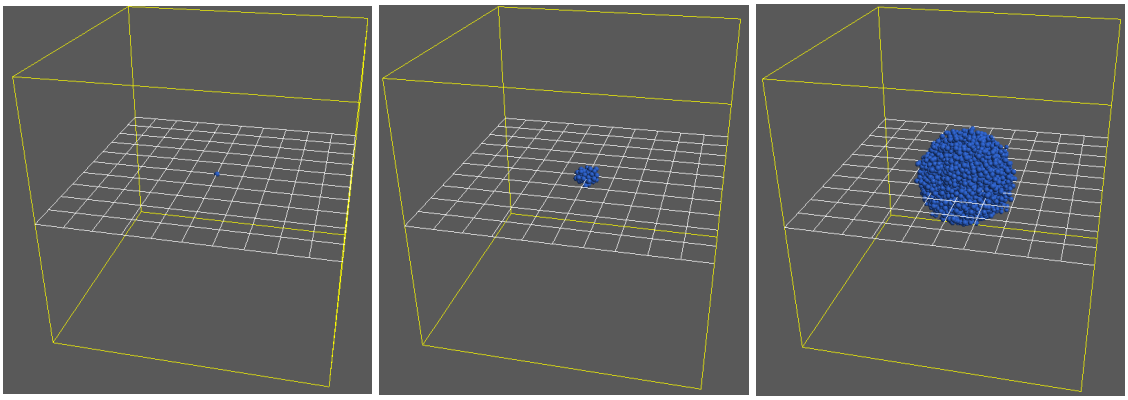
Figure 5.11: Necrotic cells volumes evolution in time. Each necrotic cell volume is plotted individuality

5.5.2 Tissue Forge

5.5.2.1 Ki-67 Basic Cycle

For the Tissue Forge Ki-67 Basic Cycle simulation we used a time-step duration of 10 min/step, as Tissue Forge is off-lattice we don't need a space conversion factor. This simulation is 3D, and we see that the cell cluster at the center (see Figure 5.12).

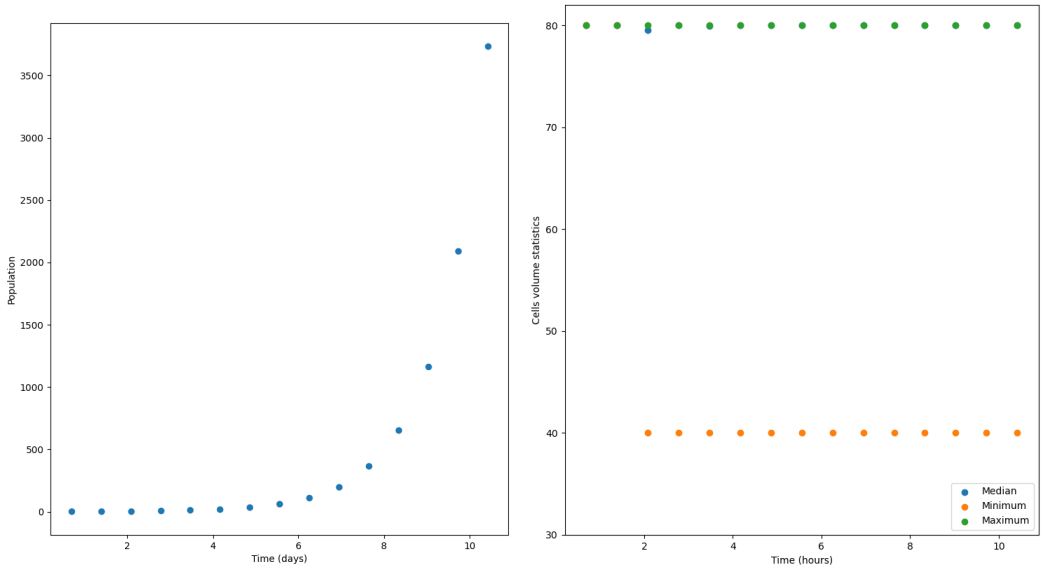
Unlike in CompuCell3D , cells in Tissue Forge are soft spheres and can reach their desired volume almost instantly. Therefore, we don't need the modification to the division transition. The median and minimum cell volume stay steady (see Figure 5.13b). The median cell volume stays close to the maximum cell volume (see Figure 5.13b) because the



(a) (b) (c)

Figure 5.12: Cells space configuration in the Tissue Forge Ki-67 Basic cycle model. a) Simulation start. b) Day 7. c) Day 15 (simulation end).

cells double in volume very fast in Tissue Forge, the cells then stay at their doubling volume until the phase duration has transpired. As before, the population growth is exponential (see Figure 5.13a).



(a) (b)

Figure 5.13: Cell population statistics for the Tissue Forge model using PhenoCellPy's Ki-67 Basic Cycle phenotype. a) Total cell population. b) Cells' volume statistic.

5.6 Discussion

The PhenoCellPy's embedded modeling package allows modelers to easily create sequences of cell behaviors and attach them to agent in agent-based models. PhenoCellPy is accessible, intuitive and enables modelers to add complexity to their model without much overhead. It implements several biological concepts, such as how to switch from one behavior to the next, how the different volumes of the cell should be modeler (*e.g.,* cytoplasmic and nuclear), and makes sure time-scales are respected. PhenoCellPy also makes the creation of new behaviors and sequences of behaviors easy.

PhenoCellPy is open-source and freely available under the BSD 3-Clause License (https://github.com/JulianoGianlupi/PhenoCellPy/blob/main/LICENSE).

5.6.1 Installation

PhenoCellPy's *alpha* version does not have any installer. To use it you should clone or download its GitHub repository [83] and add its folder to the simulation's system path. E.g., see Listing 38.

```
1 import sys
2 sys.path.extend(['C:\\PhenoCellPy_Dowload_Folder',
3 'C:/PhenoCellPy_Dowload_Folder'])
```

Listing 38: How to add PhenoCellPy to the simulation's system path.

5.6.2 Planned features

We currently have these planned features:

- Interface (API) classes for CompuCell3D and Tissue Forge
- Template interface classes

- Automatic inter-cell heterogeneity [2]
- Randomization of the initial Phase of the Phenotype
- Conda or pip distribution
- API for environment interaction (*e.g.*, detection of oxygen levels in the environment, detection of neigboring cells)
- ”Super-Phenotypes,” phenotypes made from more than one Phenotype class, and methods for switching between them

5.6.3 Requirements

PhenoCellPy only requirements, besides Python 3 support, are that NumPy and SciPy be available inside the modeling framework PhenoCellPyis embedded in.

CHAPTER 6
DISCUSSION

. I've helped the field with my cross-platform work, my translator serves as a proof of concept that it is possible to create a unified model specification for ABMs, simmilarly to what is done for population dynamics, metabolic networks, cell signaling pathways, pharmacological models, *etc.*, through SBML [12]. This discussion chapter aims to provide a overview of the findings, explore their significance, and discuss their implications in the context of the broader field of research.

Firstly, it is crucial to highlight the major findings of my research. The results obtained from the developed agent-based models elucidated the complex dynamics of COVID-19 at the cellular level, offering valuable insights into the interplay between viral infection, immune response, and treatment interventions. The models effectively captured the spatial and temporal dynamics of viral spread within cell populations and what are key parameters that can explain the variability of patient outcomes in a, naive, first infection with SARS-CoV-2 (see Chapter 2). I also successfully integrated more traditional PK-PD/PBPK models with our COVID-19 model, and indicated which questions this combination can ask that a pure PK-PD/PBPK cannot (Chapter 3). In particular, "what are the effects of cell individuality and heterogeneity on a pro-drug treatment?" I also posed some possible sources of cell individuality, *e.g.*, cell age, cell distance from capillaries. Exploring these possible sources and their precise effect can help future drug development.

The outcomes of these studies have significant implications for both the field of computational biology and the understanding of COVID-19 dynamics. By developing mechanistic models that integrate cellular behaviors, microenvironmental factors (*e.g.*, cell heterogeneity), and treatment interventions, this research contributes to a more comprehensive understanding of the disease. The insights gained from these models can inform evidence-based

decision-making in the development of therapeutic strategies, public health interventions, and drug discovery efforts.

My PhysiCell to CompuCell3D translation software is successful. The translation process successfully transferred the fundamental components of the models, while preserving key model behaviors. It showed what will be the difficult areas for a hypothetical general modeling specification for ABMs, both predicted areas and new ones. For instance, concepts that exist in one platform but not in the other (phenotypes), differences in scale limits, run time differences.

PhenoCellPy successfully implements phenotype models in a way that can be widely adopted and is easy to use. As it is a Python package it can be imported by any other Python software, and there are established methods to interface with Python from other programming languages. After we make PhysiCell available as a conda package we expect it will see widespread adoption. PhysiCell will also make models more generaly available cross-platform, as a phenotype defined using PhenoCellPy for one particular model can be imediatly used in other models in any Python-supporting platform.

In conclusion, my Ph.D. work has demonstrated the effectiveness of agent-based models in elucidating the mechanistic underpinnings of COVID-19 and anti-viral treatments. The cross-platform translation and validation of these models have paved the way for enhanced collaboration, model interoperability, and knowledge exchange within the scientific community. The findings contribute to the growing body of research on computational biology and provide a foundation for future investigations into the modeling of human health and disease that will eventually be a human-health digital twin.

6.1 Other Infrastructure Work

My work making bio-ABM more available and shareable began before building the translator and PhenoCellPy. Since the start of my Ph.D. I have been involved with nanoHUB. nanoHUB is an online platform that provides access to a wide range of nanotechnology-

related resources and tools. It is a collaborative effort that aims to facilitate research, education, and collaboration in the field of nanotechnology. nanoHUB provides a platform for researchers to share their work, users can upload and publish their own tools, resources, and research findings, fostering knowledge exchange and collaboration. This is specially useful for journal publication of models. Instead of asking the referees and reader to download your model and the software it runs in, you can deploy the model online and link to it from the publication. nanoHUB is also useful as an educational tool, educational demos can be hosted there and used in the classroom.

More precisely, I am involved with nanoBIO, nanoBIO is a project that aims to extend nanoHUB from being just about nanotechnology to also include biological models. I have helped the deployment of CompuCell3D on nanoHUB, and I built a helper script that helps prepare a CompuCell3D model for deployment on nanoHUB [92] (see online: `https://github.com/JulianoGianlupi/cc3d-nanoHub-tool-maker`), as well as a template for the deployment of tellurium [93] models [94] (see online: `https://github.com/JulianoGianlupi/tellurium-nanohub-base`). I have deployed 17 educational tools [94–111].

6.2 Future work

6.2.1 Translator

As mentioned, James Glazier and Paul Macklin were awarded a NSF POSE grant to build a standardized ecosystem for virtual tissue modeling. My efforts in building a translator will play a pivotal role in the success of this effort. I've pointed out areas that will be a challenge and addressed some of them. These challenging areas will most likely continue to be difficult when addressing other platforms beyond CompuCell3D and PhysiCell. I believe the main areas of difficulty won't be related to forces or how behaviors are defined, but on concepts that one framework has an the other doesn't. In my work translating PhysiCell into CompuCell3D this was the phenotypes, this was such a big hurdle that it became its

own project (PhenoCellPy).

Near future developments for the translator will focus on finding an adequate form for the cell-cell contact energy in CompuCell3D that reproduces the adhesion-repulsion forces of PhysiCell. Another area of focus will be on improving cell movement in the translated simulation (*i.e.*, implementing a version of PhysiCell's bias), and translating the initial cell layout used by PhysiCell.

6.2.2 PhenoCellPy

PhenoCellPy is a promising Python package that will help computational biologists build ABMs for cellular systems. To make its adoption easy the next step in its development is creating a Conda or pip distribution package for PhenoCellPy. That will make PhenoCellPya part of the user's python environment and importing it will be the same as importing any other python package (*e.g.*, NumPy).

Next we will focus on more robust API features, coordinating the embedded PhenoCellPy model with the main model will be easier and faster. The API will also make the integration of PhenoCellPy with other modeling platforms beyond CompuCell3D and TissueForge straight forwards.

6.2.3 My Future Post-Doctoral Research

Besides continuing the development of PhenoCellPy and advising on the ABM definition standard, I will begging a new research line with Dr. Amber Smith at UTHSC.

I will work on building models of pneumococcus pneumonia infection. How the pneumococci bacteria avoid detection by the immune system, change phenotype, how the disease progresses, and so on.

Appendices

APPENDIX A

MULTISCALE MODEL OF ANTIVIRAL TIMING, POTENCY, AND HETEROGENEITY EFFECTS ON AN EPITHELIAL TISSUE PATCH INFECTED BY SARS-COV-2

A.1 Simple PK model for Remdesivir and GS-443902

Gallo [33] developed a model to predict the intracellular concentration of remdesivir triphosphate (GS–443902) after IV infusion of remdesivir in humans. Their model is based on measurement of GS–443902 in peripheral blood mononuclear cells (PBMC) following IV infusion of remdesivir. That model is then used as a surrogate for predicting the concentration in lung cells assuming the exposure, uptake and metabolism of the two cell types are similar. Here we develop a simpler model designed to just predict the GS–443902 concentration in lung cells based on the data and model of Gallo, along with data from Humeniuk *et al.* [23], and with additional PBMC data from the Gilead application to the European Medicines Agency's for the compassionate use of remdesivir to treat COVID-19 patients [112]. The goal of this model is simply to estimate reasonable tissue concentrations of GS–443902 versus time as a function of remdesivir dose in repetitive dosing scenarios. These dosing scenarios are like those used by Gallo and in the compassionate use document. The model layout is shown in Figure 3.1 (in the main paper) and the ordinary differential equations,

$$\frac{\mathrm{d}(C_{GS} \cdot vol)}{\mathrm{d}t} = \left(\frac{D_{rmd} \cdot k_{in}}{\tau_{\mathrm{I}}}\right) - vol \cdot k_{out} \cdot C_{GS} \ , \tag{A.1}$$

$$\frac{\mathrm{d}(C_{GS}^{AUC})}{\mathrm{d}t} = C_{GS} \ . \tag{A.2}$$

Differential equations for the minimized remdesivir PK model. Note that vol in Equations A.1 and A.2, in Figure 3.1 and in Table 3.1, and is the effective volume of the single compartment in the model. Here, D_{rmd} is in mols. The model's input for remdesivir is in milligrams, which the COPASI code converts to moles (*e.g.*, "Remdes_dose_mol") so that conversion of remdesivir to GS–443902 is one-to-one.

Our simplified model was constructed in COPASI (http://copasi.org), an SBML compliant biochemical system simulator well suited for modeling chemical and biological processes represented by ordinary differential equations (ODEs). COPASI allows for easy model definition, including the use of links to biological ontologies for unique identification of species and concepts, as well as a suite of tools for running simulations, parameter fitting, sensitivity analysis etc. The COPASI file (which is setup to do the parameter estimation task) along with the parameter files based on Gallo, Humeniuk *et al.* and the compassionate use document are available at https://github.com/JulianoGianlupi/covid-tissue-response-models/, or the entire repository can be downloaded from https://github.com/JulianoGianlupi/covid-tissue-response-models/zipball/fossilized-repo/. Table A.1 summarizes the parameters and measurements from the Humeniuk study and compassionate use documents that we used to calibrate our model.

Table A.1: Data used to calibrate the simple remdesivir to GS–443902 prediction model. *Humeniuk's Table 4 in [23]; **EU Compassionate Use's Table 16 in [112]; *** Infusion duration not given, assumed to be 1 hour.

Parameter	**Unit**	**Humeniuk* Cohort 7**	**Humeniuk* Cohort 8**	**EU Compassionate Use****
Remdesivir Dose	mg	150	75	200
Infusion Duration	h	2	0.5	1.0 ***
GS–443902 C_{max}	uM	6.0	5.9	9.8
GS–443902 C_{24hr}	uM	3.7	3.3	6.9
GS–443902 $t_{1/2}$	1/h	36	49	
GS–443902 AUC_{24h}	h*uM			157.4
GS–443902 AUC_{inf}	h*uM	297	394	
GS–443902 AUC_{last}	h*uM	272	340	

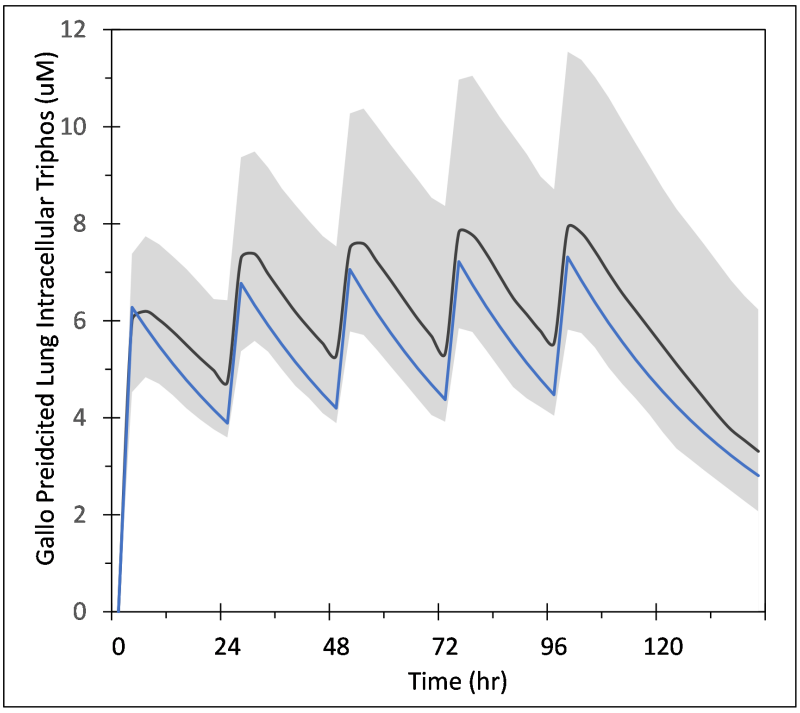


Figure A.1: Comparison of our simplified model to Gallo's population simulation plot of predicted intracellular lung GS–443902 concentration. Gallo's mean response is shown by the black line and the 95% interval is shaded. This data is from 5000 simulations in which the two key parameters in the Gallo model were sampled from distributions with 20% CV. See Gallo and their supplement Figure S5 for more information. Blue line is our simplified model's output. The Gallo data used in this plot was digitized from the publication using `https://automeris.io/WebPlotDigitizer/`.

A.1.1 COPASI Codes

The COPASI model files are available on GitHub as described earlier. Simulations were developed in COPASI version 4.30 (Build 240) and have been checked through version version 4.34 (Build 251). Below we give details for running the two models.

A.1.1.1 COPASI model file for parameter fitting

The file `GS-443902_PBMC_PK_v05_3data_plusEurope.cps` does the parameter estimation task based on the data of [23]. The data files are included in the GitHub repository. This COPASI file is set up to do both the parameter fitting task and a basic time course simulation. Load the file in COPASI and insure that the following data files are in the same

folder that contains the COPASI .cps file:

- `Humeniuk_PK_data_Europe_Table_16.txt`
- `Humeniuk_PK_data_Table_4_Cohort7.txt`
- `Humeniuk_PK_data_Table_4_Cohort8.txt`

The COPASI model implements the infusion period with an event named "Infusion", which changes the variable "k_{in}" from 1 to 0 when the model's time exceeds the length of the "τ_I" parameter (see Equation A.1 and the COPASI code). Several other events are used to determine the T_{max}, C_{max}, AUC at 24 hours etc. The remdesivir dose is input in milligrams and converted to moles by the COPASI code. To run the parameter estimation, select "Task", "Parameter Estimation", then select an estimation method. We used the "Particle Swarm" method with the default values. In the Upper right of the COPASI window the button "Experimental Data" will open the data window. If the data files listed above are in the same directory as the .cps file then this should be populated with the data mappings for each of the three files. These values are also summarized in Table A.1. In the upper right of the COPASI window click the "update model" checkbox so the fitted parameters are updated into the starting values for the COPASI model. Click "Run" and the parameter are estimated and available on the "Results" sub-item. The Particle Swarm should iterate down to an objective value of about $6.2x10^{-11}$. The models has now been updated with the results for the terminal clearance half life and the effective compartment volume, typically 30.2 hours and 38.4 liters, respectively. A time course can be run using these parameters by selecting "Tasks", "Time Course" then "Run". This COPASI model generate several graphs, some of which are for the fitting task and some for the time course task.

A.1.1.2 COPASI model file for calculating GS–443902 from repetitive remdesivir doses

The file `GS-443902_PBMC_PK_v05_3data_repeatDose_Gallo.cps` simulates repetitive doses of 200, then 100x4mg/day. This model is based on the parameter

fitting model described above. The variable k_{in} controls the infusion periods as well as the infusion dose since it can be thought of as a multiplier on the infusion dose (see Equation A.1). During the initial infusion k_{in} is 1 and in subsequent infusion periods 0.5 to implement the initial 2x loading dose. This factor in combination with the remdesivir dose (Remdes_dose_mg) value gives the 200mg then four times 100 mg dosing pattern. The infusion timing is shown in the "k_{in}" plot generated by the COPASI code. Loading this model into COPASI, then "Task", "Time Course", "Run" produces output similar to what is shown in Figure A.1. Note that the maximum units in COPASI on the Y-axis are mole/liter with values near $10^{-5}mol/L$, which corresponds to the $\sim 10\mu M$ values in Figure A.1.

A.2 Table of parameters from Sego *et. al*

Here we present all the parameters from Sego *et al.* [1]. In this work, we only changed the simulation step time-length, going from 1200s to 300s. We have marked it with an asterix and included our value in parenthesis. The reasoning for the choice of each of the parameters can be found in their original paper in Table 1.

Table A.2: Sego et al.'s conversion factors

Conversion Factors	**Value**
Simulation step Δt	$1200.0s$ * ($300.0s$)
Lattice width	$4.0\mu m$
Scale factor for concentration	$10^{-14}mol$

Table A.3: Sego et al.'s parameters part 1

Simulation parameter name	Value	Simulation parameter name	Value
Cell diameter	$12.0\mu m$	Viral decay rate γ_{vir}	$7.71 \times 10^{-6} s^{-1}$
Replication rate r_{max}	$(1/12)10^{-3} s^{-1}$	Cytokine diffusion coefficient D_{cyt}	$0.16\mu m^2 s^{-1}$
Translating rate r_t	$(1/18)10^{-3} s^{-1}$	Cytokine diffusion length λ_{cyt}	$100\mu m$
Unpacking rate r_u	$(1/6)10^{-3} s^{-1}$	Cytokine decay rate γ_{cyt}	$1.32 \times 10^{-5} s^{-1}$
Packaging rate r_p	$(1/6)10^{-3} s^{-1}$	Maximum cytokine immune secretion rate $\sigma_{cyt}(immuneactivated)$	$3.5 \times 10^{-4} pM s^{-1}$
Release rate r_s	$(1/6)10^{-3} s^{-1}$	Immune secretion midpoint $V_{cyt}(immuneactivated)$	$1pM$
Scale factor for number of mRNA per infected cell $mRNA_{avg}$	$1000 cell^{-1}$	Cytokine immune uptake rate $\omega_{cyt}(immuneactivated)$	$3.5 \times 10^{-4} pM s^{-1}$
Viral dissociation coefficient r_{half}	2000	Maximum cytokine infected cell secretion rate $\sigma_{cyt}(infected)$	$3.5 \times 10^{-3} pM s^{-1}$
Viral diffusion coefficient D_{vir}	$0.01\mu m^2 s^{-1}$	Infected cell cytokine secretion mid-point $V_{cyt}(infected)$, $V_{cyt}(virus - releasing)$	0.1
Viral diffusion length λ_{vir}	$36\mu m$	Cytokine secretion Hill coefficient h_{cyt}	2

Table A.4: Sego et al.'s parameters part 2

Simulation parameter name	Value
Immune cell cytokine activation $EC_{50,cyt,ac}$	$10pM$
Immune cell equilibrium bound cytokine EQ_{ck}	$210pM$
Immune cell bound cytokine memory ρ_{cyt}	$0.99998s^{-1}$
Immune cell activated time	$10h$
Oxidation Agent diffusion coefficient D_{oxi}	$0.64\mu m^2 s^{-1}$
Oxidation Agent diffusion length λ_{oxi}	$36\mu m$
Oxidation Agent decay rate γ_{oxi}	$1.32 \times 10^{-5} s^{-1}$
Immune cell oxidation agent secretion rate σ_{oxi}	$3.5 \times 10^{-3} pM\ s^{-1}$
Immune cell C_{cyt} threshold for Oxidation Agent release t_{sec}	$10 A.U. = 1.5625pM$
Tissue cell C_{oxi} threshold for death $t_{death-oxi}$	$1.5 A.U. = 0.234375pM$
Initial density of unbound cell surface receptors R_o	$200cell^{-1}$

Simulation parameter name	Value
Virally-induced apoptosis dissociation coefficient V_{apo}	100
Virally-induced apoptosis characteristic time constant α_{apo}	$20min$
Immune cell activation Hill coefficient h_{act}	2
Immune response add immune cell coefficient β_{add}	$1/1200s^{-1}$
Immune response subtract immune cell coefficient β_{sub}	$1/6000cell^{-1}s^{-1}$
Immune response delay coefficient β_{delay}	$1.2 \times 10^6 s$
Immune response decay coefficient β_{decay}	$1/12000s^{-1}$
Immune response cytokine transmission coefficient α_{sig}	0.5
Immune response probability scaling coefficient α_{immune}	0.01
Number of immune cell seeding samples $n_{seeding}$	10
Initial immune cell target volume	$64\mu m^3$

Table A.5: Sego et al.'s parameters part 3

Virus-receptor association affinity k_{on}	$1.4 \times 10^4 M^{-1}\ s^{-1}$
Virus-receptor dissociation affinity k_{off}	$1.4 \times 10^{-4} s$
Infection threshold	1
Uptake Hill coefficient k_{upt}	2
Uptake characteristic time constant α_{upt}	$20min$
Virally-induced apoptosis Hill coefficient h_{apo}	2
Immune cell lambda volume λ_{volume}	9
Initial number of immune cells	0
Immune cells lambda chemotaxis $\lambda_{chemotaxis}$	1
Intrinsic Random Motility H^*	10
Contact coefficients J (all interfaces)	10

A.3 Quantitative metrics of treatment outcome

The classification of treatment outcomes into fast clearance, slow clearance, partial containment, or widespread infection is done by using quantitative metrics in an algorithm. We always use the median measurements of the simulation replicas for each parameter combination. We first look at the median time course of the uninfected population, second at the median time course of the extracellular viral load, third at the extracellular virus AUC from treatment start.

If the median uninfected population at simulation end is below ten or less than half the uninfected population at the start of treatment we classify it as ineffective with widespread infection.

The next metric we look at is median extracellular viral load. If the viral load goes below a threshold of $1.3A.U.$ the simulation has cleared the virus at least once (there may be subsequent releases of extracellular virus) but we still don't know if the infection was contained or not. If the extracellular virus was cleared in less than 14 days of treatment and the extracellular virus concentration does not rise above a slightly higher threshold of $1.1 \times 1.3A.U. = 1.43A.U.$ after treatment initiation we classify the treatment as effective with fast containment.

If the virus is cleared but then there is a rise above $1.43A.U.$ we look at the maxima and minima of the logarithm of extracellular viral load post 14 days of treatment. We calculate the difference of the last maxima and the first minima (ΔM) and we compare it to another threshold of $10^{-4}A.U.$ If the difference is close to zero, $|\Delta M| < 10^{-4}A.U.$, we classify the treatment as ineffective with a partial containment. If the difference is less than the negative of the threshold ($\Delta M < -10^{-4}A.U.$) we classify the treatment as effective and the clearance as slow. If the difference is above the positive threshold ($\Delta M > 10^{-4}A.U.$) we classify the result as widespread infection.

If the extracellular virus level never goes below the $1.3A.U.$ threshold we first check

against the $1.43 A.U.$ threshold, if the levels have gone below it we classify the treatment as partial containment. If not we look at ΔM (as before) and at the AUC from treatment initiation (AUC_{TI}). If AUC_{TI} is below $300 A.U.$ we classify the treatment as ineffective with a partial containment. If not we use ΔM and the threshold of $10^{-4} A.U.$ for the classification.

A.4 Instructions for running the multiscale CompuCell3D simulations and for analyzing the results

The simulation code as well as the scripts used to analyze the results are hosted in gitHub at https://github.com/JulianoGianlupi/covid-tissue-response-models/, download repository.

A.4.1 CompuCell3D simulations

To run the ABM you can use either your personal computer or a cluster. You need to download CompuCell3D version 4.2.3 (or newer, investigations were done in 4.2.3), `https://compucell3d.org/SrcBin`. For local installations, run the python script `cellular-model/batch_run.py`, and (optionally) define the output directory in the script. For cluster execution, change the output directory in `cellular-model/batch_exec.py` and run the script `cellular-model/batch_exec.sh`. The script is set up for Slurm scheduling systems. In those files you can define the output directory (variable `sweep_output_folder`).

All parameters that were varied and investigated are in the file `cellular-model/investigation_dictionaries.py` and are imported to `batch_run.py` and `batch_exec.py`. To change the investigated parameters, change the dictionary used as `mult_dict` in one of those files. *e.g.*:

- `mult_dict = treatment_starts_0`, parameters varied in the fine investigation with treatment starting with the infection of 10 epithelial cells

- `mult_dict = treatment_starts_3_halved_half_life`, parameters varied in the fine investigation with treatment starting3 days post the infection of 10 epithelial cells and with the half life of GS–443902 halved.

Those files also support defining how many replicas to be executed for each parameter combinations to do with the variable `num_rep`. The workflow will then run through all combinations of parameters in mult_dict, generating and running "`num_rep`" simulations for each. All results will be stored in "`sweep_output_folder`". Each parameter combination is called a "set" and all results of a set are in their respective folder. *e.g.*, for the following parameter dictionary

```
treatment_starts_0 = {
    'first_dose': [0],
    'dose_interval': [1, 1.5, 2, 2.5, 3,
        3.5, 4, 4.5, 5, 5.5, 6],
    'ic50_multiplier': [0.01, 0.02, 0.03, 0.04,
        0.05, 0.06, 0.07, 0.08,
        0.09, 0.1],
    't_half_mult': [1]}
```

the first set (`set_0`) uses `first_dose=0`, `dose_interval=1`, `ic50_multiplier=0.01`, and `t_half_mult=1`. The results for it will be in `sweep_output_folder/set_0`, each replica will be in `sweep_output_folder/set_0/run_0`, `sweep_output_folder/set_0/run_1`, [...], `sweep_output_folder/set_0/run_num_rep`. The second set (`set_1`) then uses `dose_interval=1.5` and `ic50_multiplier=0.01`; and so on.

A.4.2 Results analysis

There are two steps for the analysis of the ABM results, in the first step the median behaviors of the simulation replicas are measured and used for classifying the treatment among the classes defined in the methods section 3.2.5. The second step is to, then, generate the parameter variation Figures (*e.g.* Figure 3.6b).

The file `cellular-model/grid_color_picker.py` does step one. You only need to set the variable `base_path` to your path to the "set" folders. Then step two is performed by `cellular-model/PostMultiSet.py`, change `base_path` in it to be the same as in step one.

A.5 Supplementary results for the untreated simulations with different initial conditions

Here we have the results for the untreated simulations with different initial conditions, namely with 1, 2, 5 and 10 infected cells at the start of the simulation. All of the results for this appendix use 400 simulation replicas for each set of parameters used. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue.

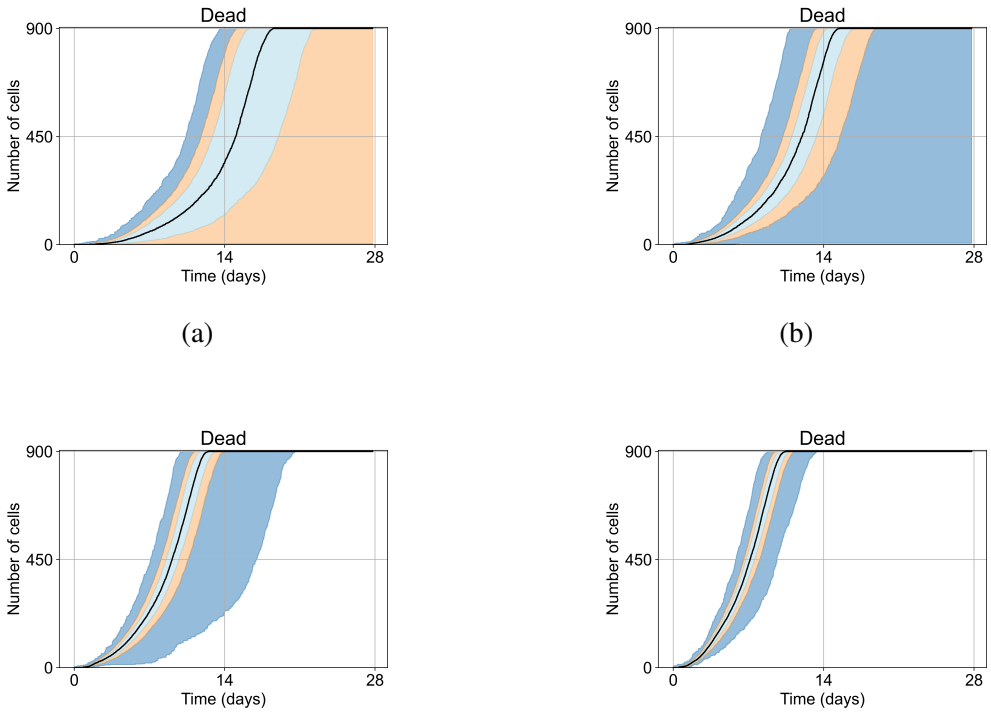


Figure A.2: Dead cell populations for 400 replicas of Sego *et al.*'s model [1]. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.2a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.2b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.2c) Simulations start with 5 initially infected cells. A.2d) Simulations start with 10 initially infected cells.

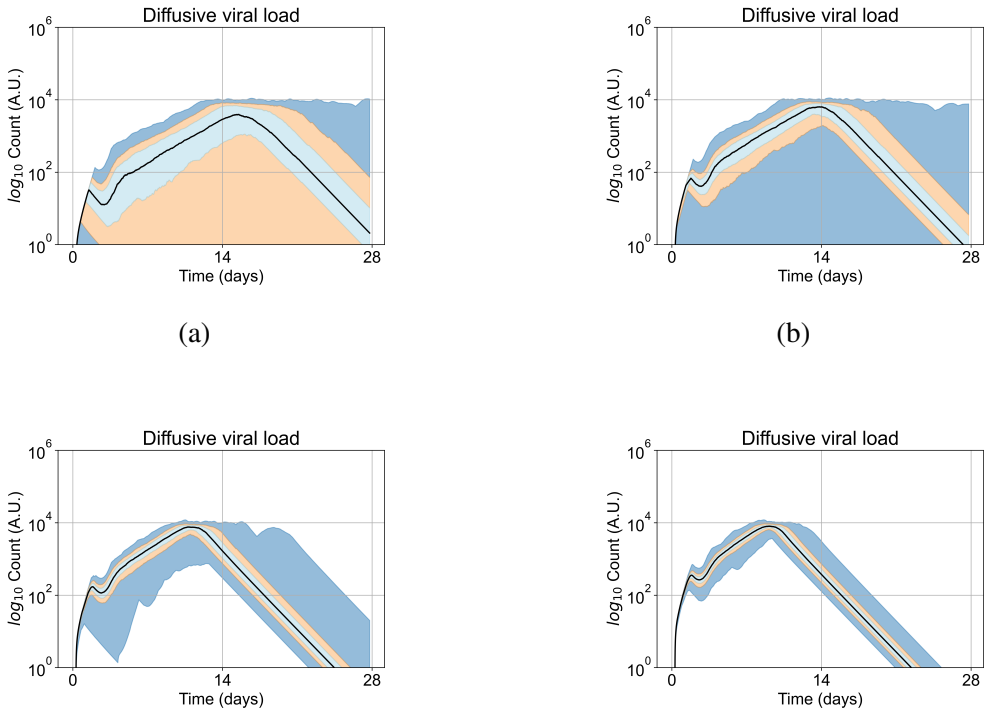


Figure A.3: Extracellular viral load for 400 replicas of Sego *et al.*'s model [1], y-axis in log scale. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.3a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.3b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.3c) Simulations start with 5 initially infected cells. A.3d) Simulations start with 10 initially infected cells.

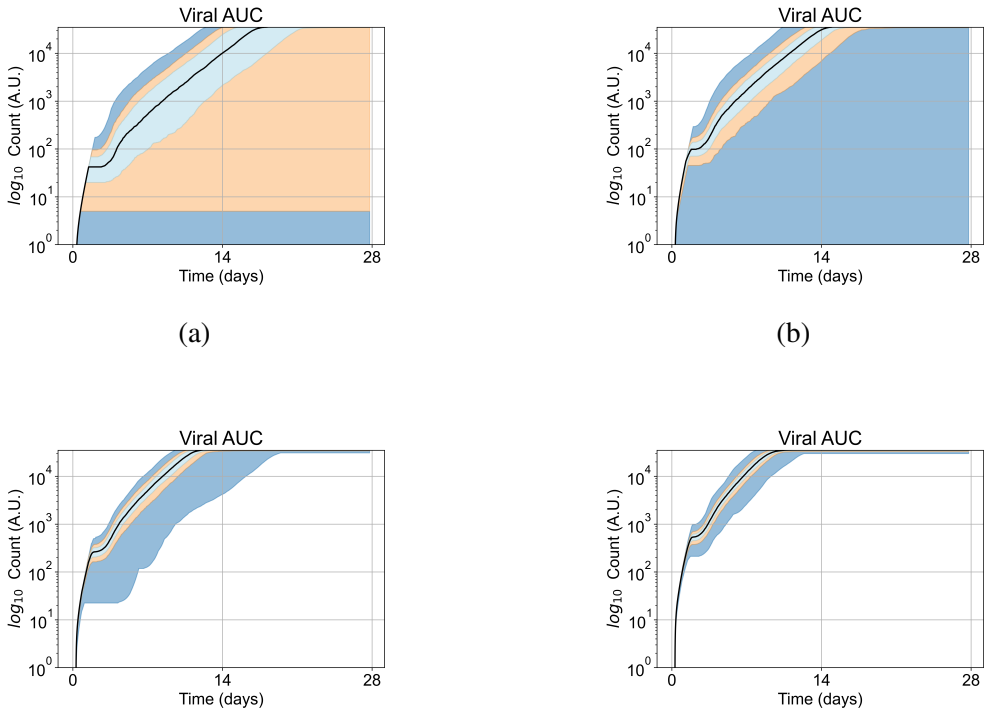


Figure A.4: Extracellular viral AUC for 400 replicas of Sego *et al.*'s model [1], y-axis in log scale. In all the cases the medians of simulation replicas are in black lines, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and the 25th to 75th as light blue. A.4a) Simulations start with 1 initially infected cell and 7 simulations result in failure to infect (1.75% of replicas), the 90th quantile includes the upper bound of the number of cells. A.4b) Simulations start with 2 initially infected cells where 5 simulations result in failure to infect (1.25%), the 100th quantile includes the upper bound of the number of cells. A.4c) Simulations start with 5 initially infected cells. A.4d) Simulations start with 10 initially infected cells.

A.6 Supplementary results from treatment initiation delay, antiviral potency, and GS-443902 half-life variation

All of the results for this appendix use 8 simulation replicas for each set of parameters used. For all subfigures the median measurement of simulation replicas is the black line, the 0th to 100th quantiles are shaded as dark blue, 10th to 90th shaded in orange, and 25th to 75th as light blue. The subplots with axis colored in green are classified as repid clearance, in blue as slow clearance, in black partial containment, in red widespread infection. The sections

names describe if the results in it are for homogeneous or heterogeneous metabolism, if the half-life of GS–443902 was altered, and how long is the delay from the infection of ten epithelial cells to treatment initiation.

A.6.1 Homogeneous metabolism, regular GS–443902 half-life

A.6.1.1 Treatment initiation with infection of ten epithelial cells

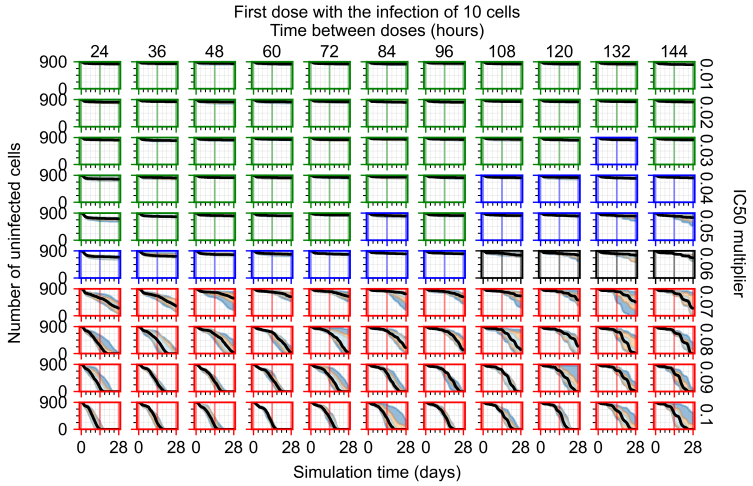


Figure A.5: Uninfected population.

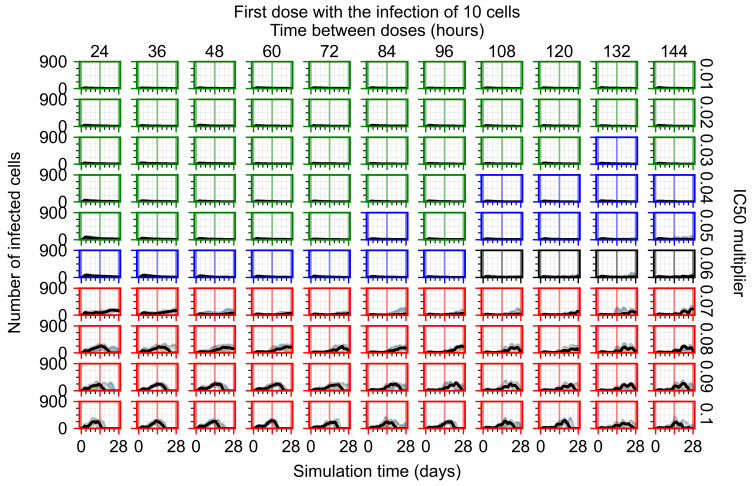


Figure A.6: Infected (eclipse phase) population.

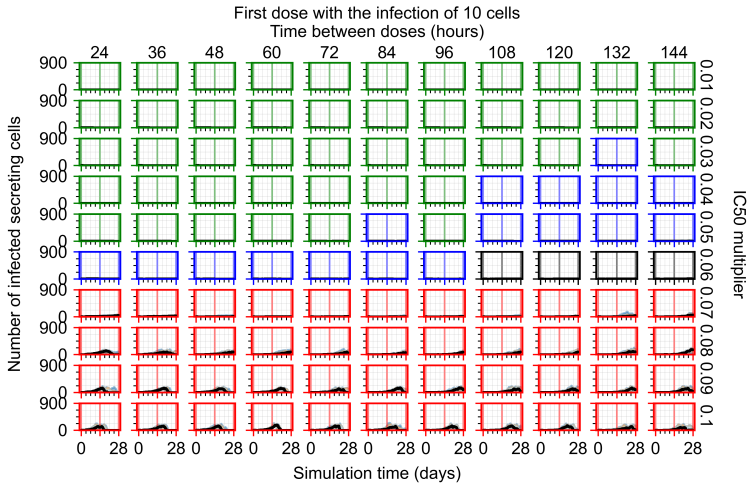


Figure A.7: Infected (secreting extracellular virus) population.

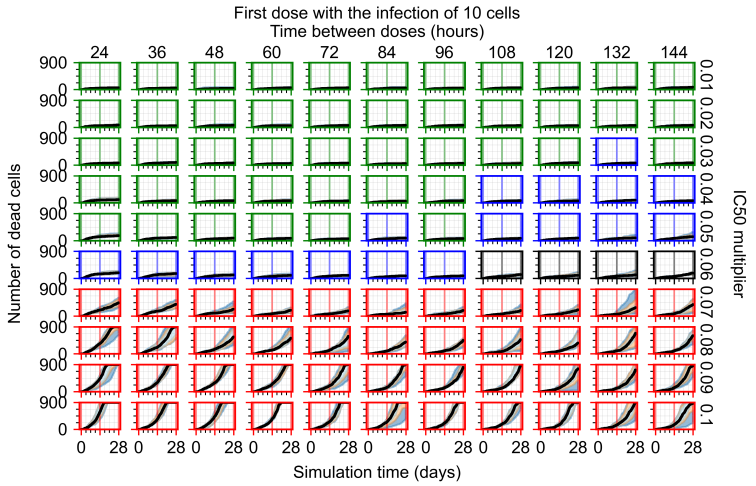


Figure A.8: Dead population.

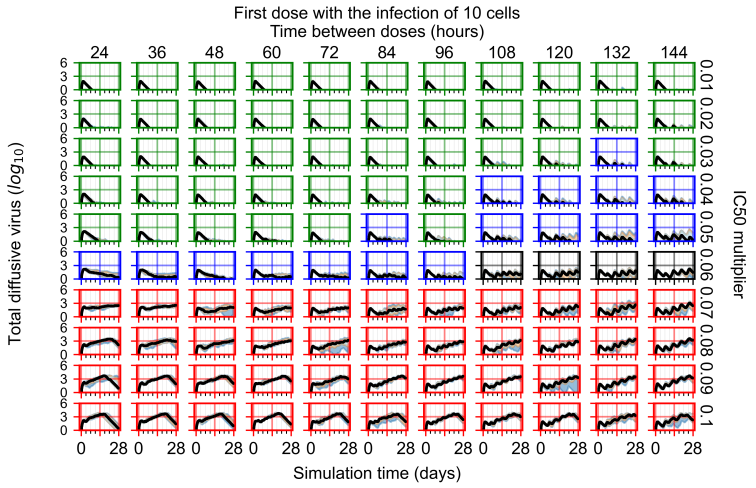


Figure A.9: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

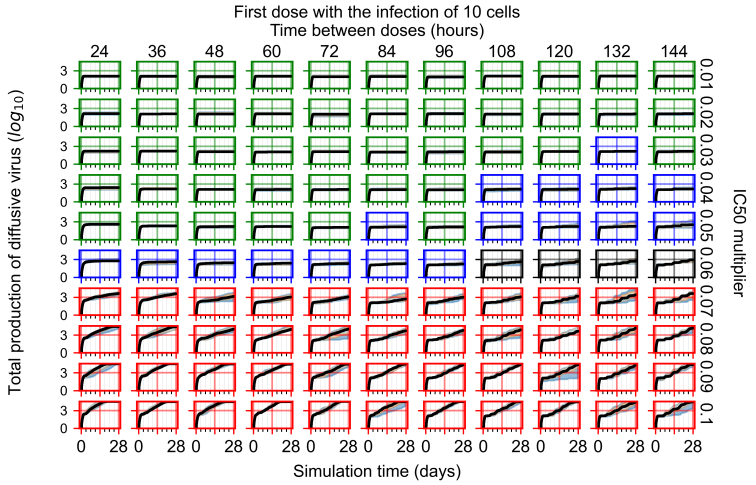


Figure A.10: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

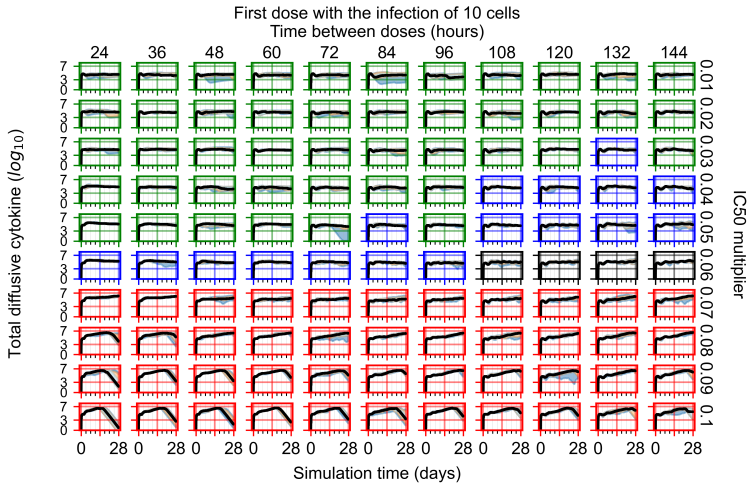


Figure A.11: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

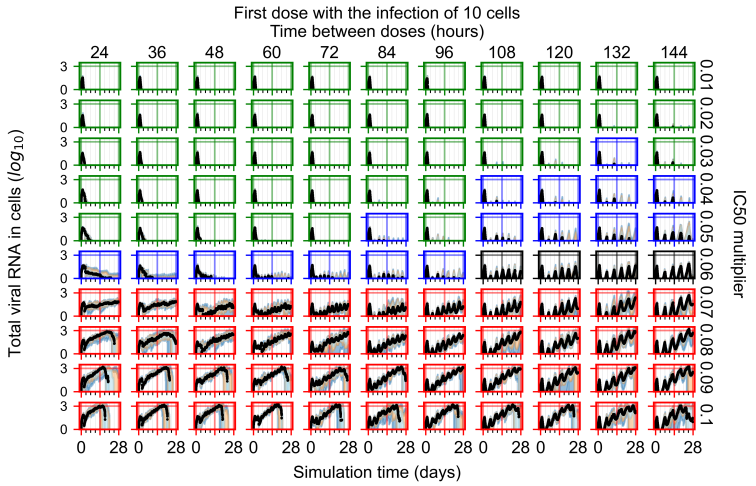


Figure A.12: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

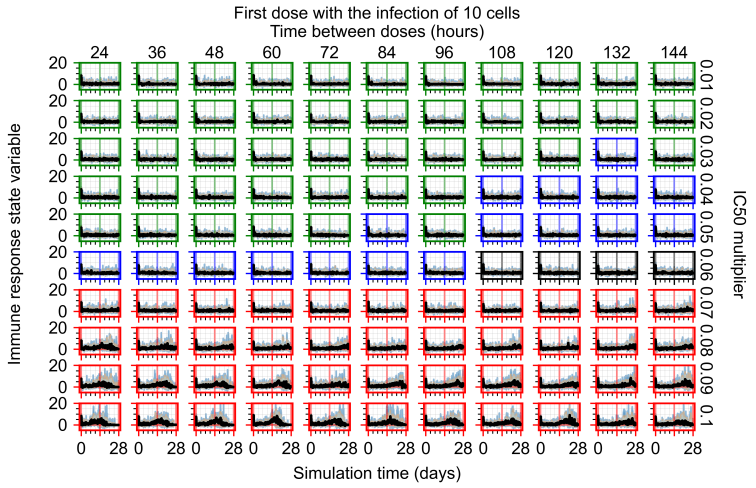


Figure A.13: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.1.2 Treatment initiation 12 hours post the infection of ten epithelial cells

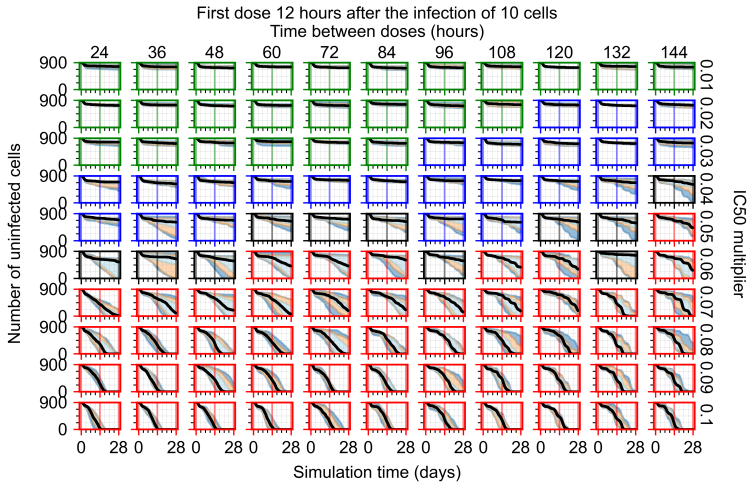


Figure A.14: Uninfected population.

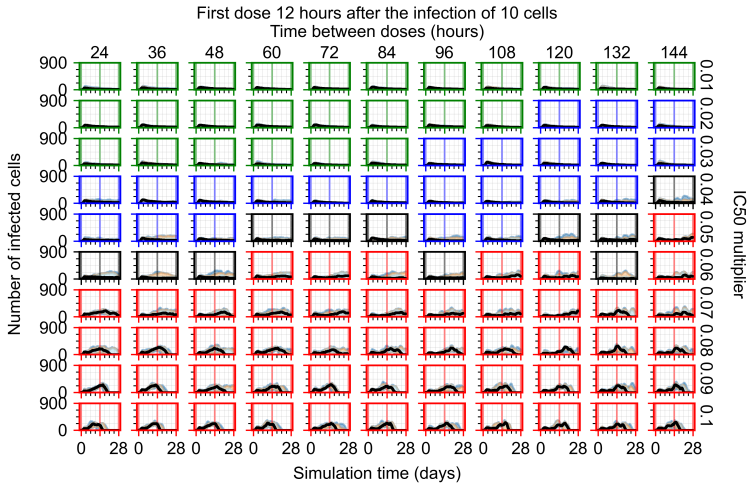


Figure A.15: Infected (eclipse phase) population.

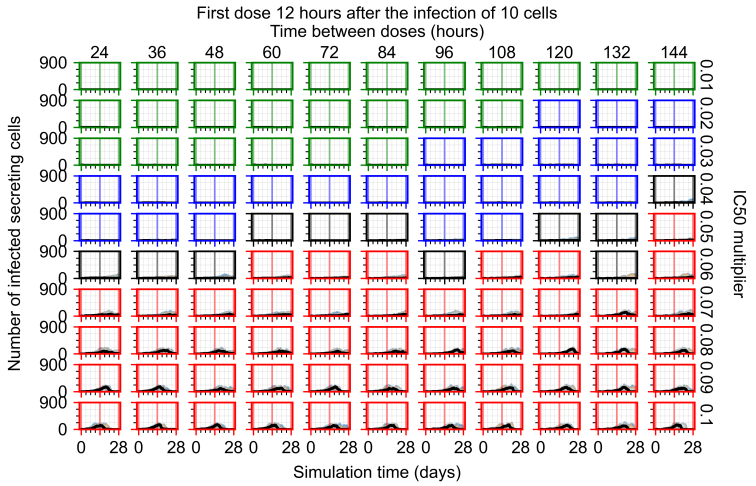


Figure A.16: Infected (secreting extracellular virus) population.

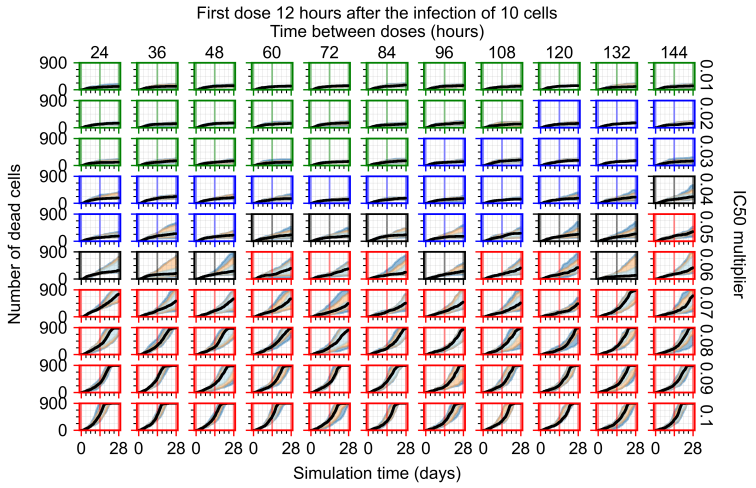


Figure A.17: Dead population.

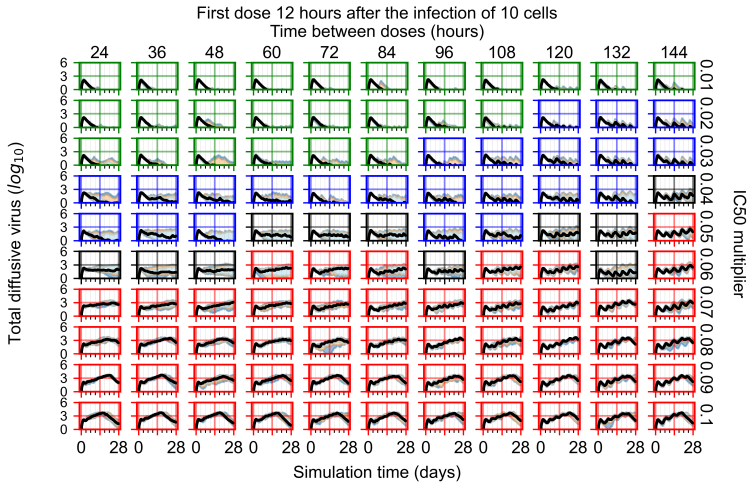


Figure A.18: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

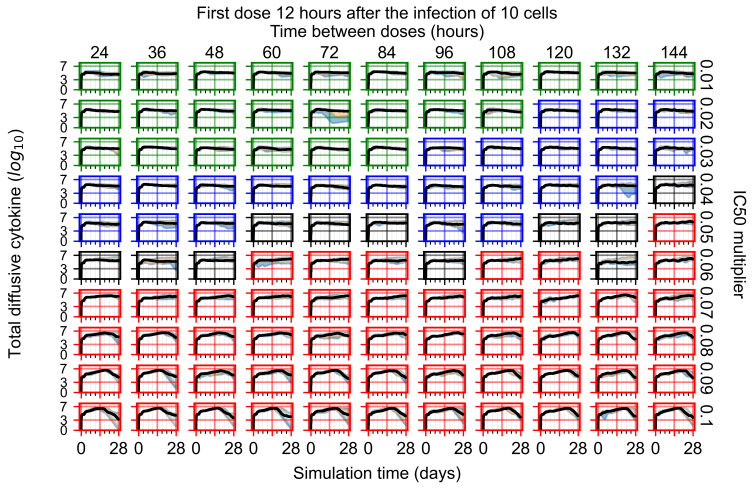


Figure A.19: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

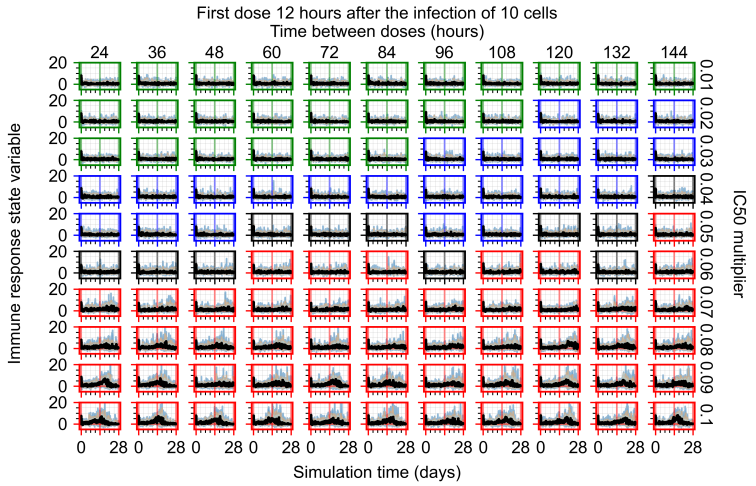


Figure A.20: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.1.3 Treatment initiation one day post the infection of ten epithelial cells

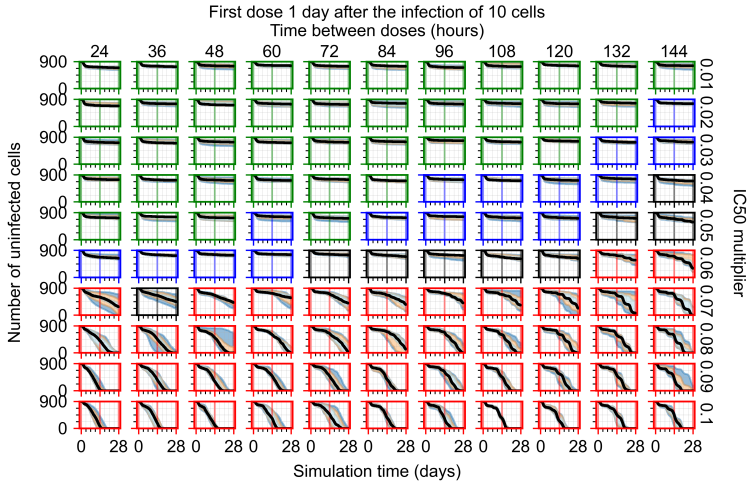


Figure A.21: Uninfected population.

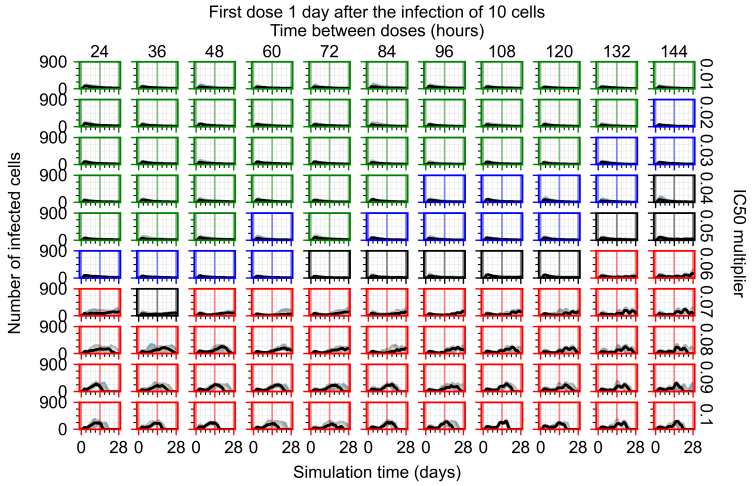


Figure A.22: Infected (eclipse phase) population.

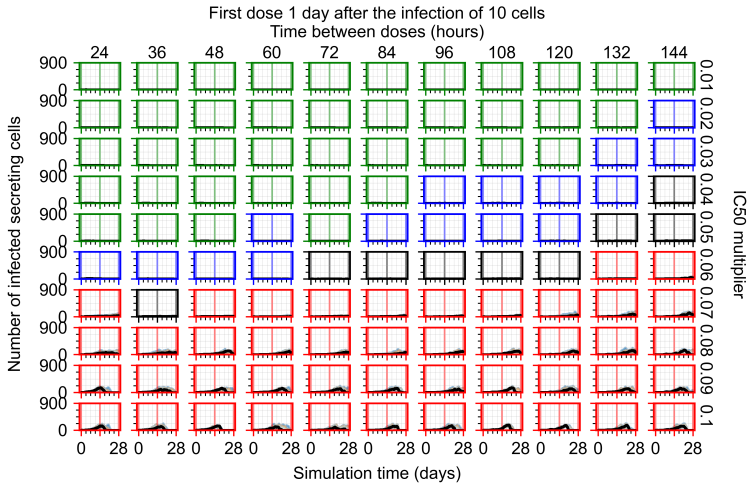


Figure A.23: Infected (secreting extracellular virus) population.

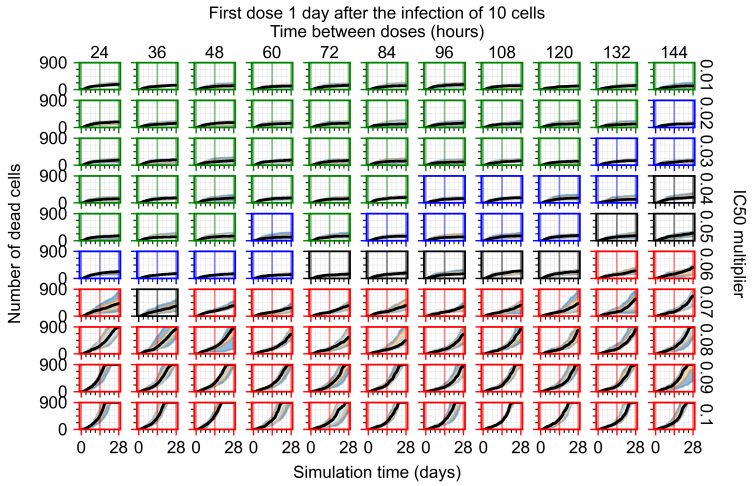


Figure A.24: Dead population.

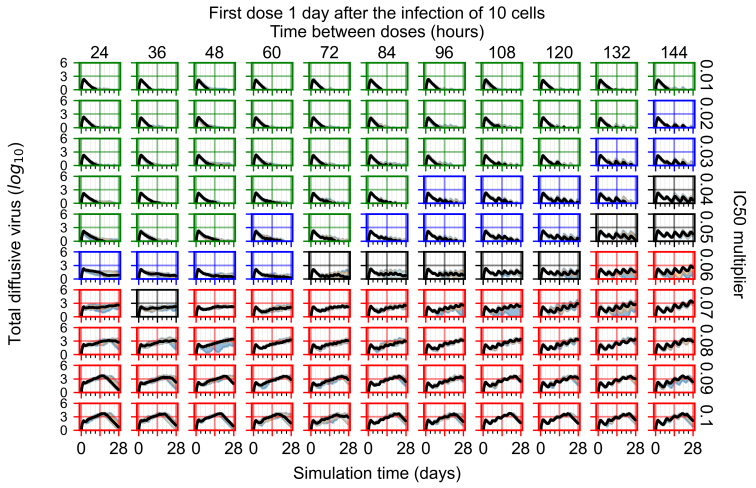


Figure A.25: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

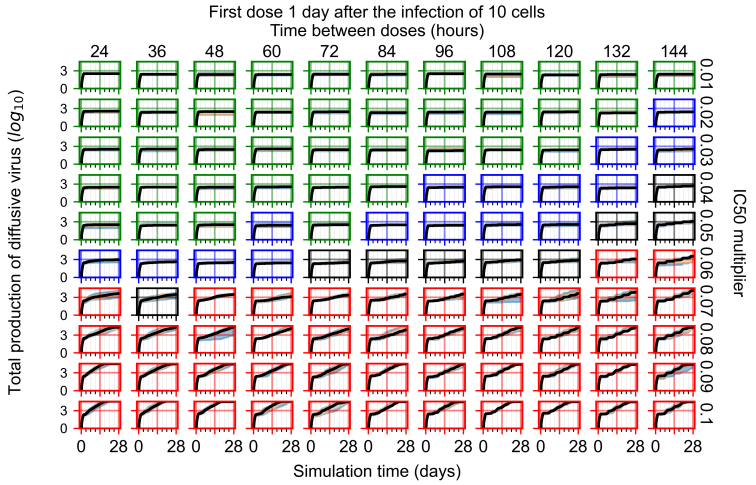


Figure A.26: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

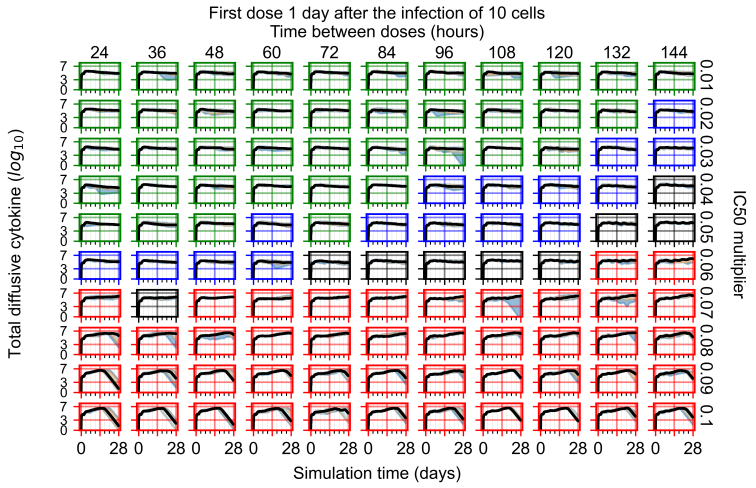


Figure A.27: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

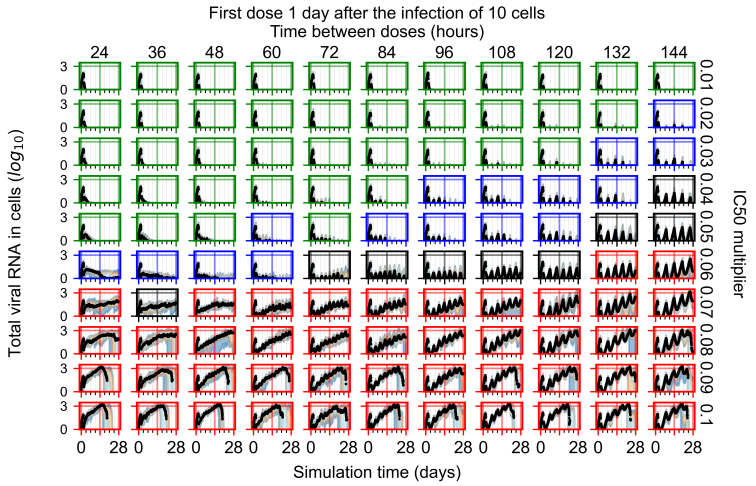


Figure A.28: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

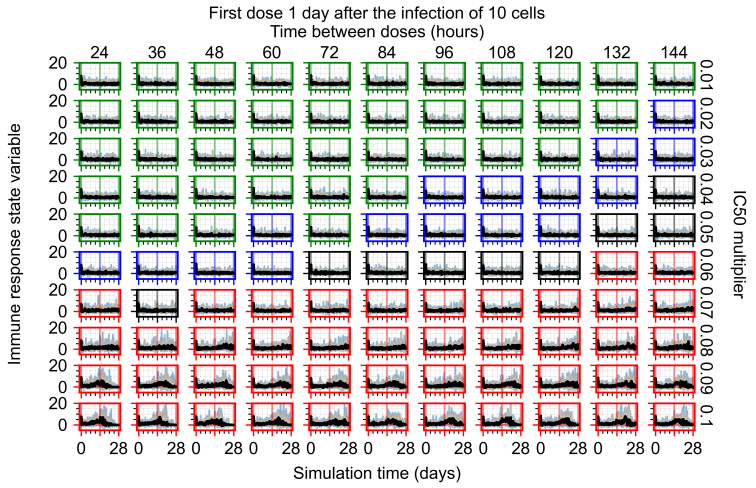


Figure A.29: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.1.4 Treatment initiation three days post the infection of ten epithelial cells

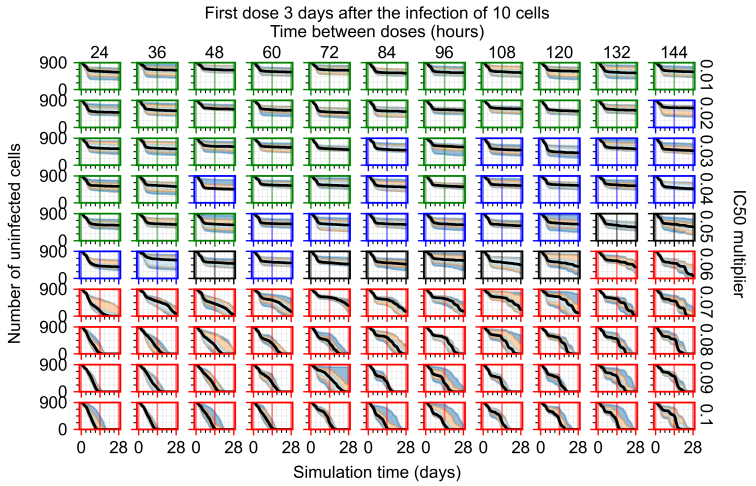


Figure A.30: Uninfected population.

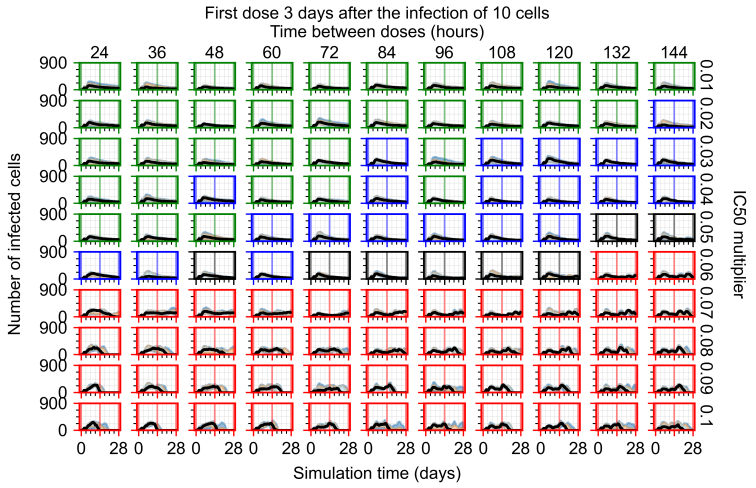


Figure A.31: Infected (eclipse phase) population.

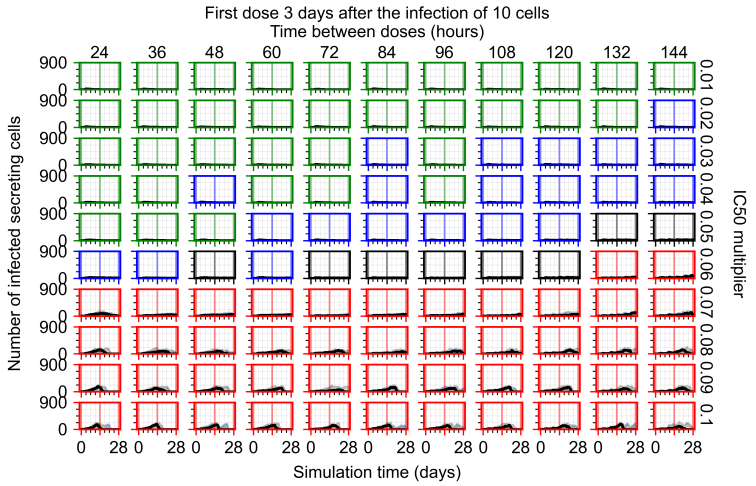


Figure A.32: Infected (secreting extracellular virus) population.

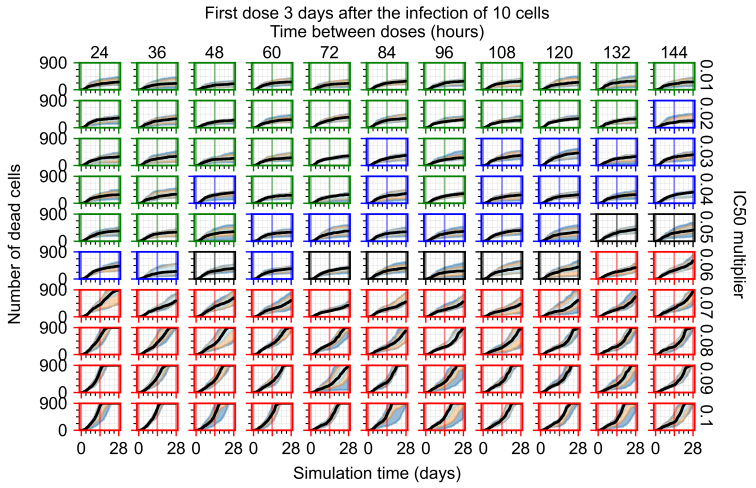


Figure A.33: Dead population.

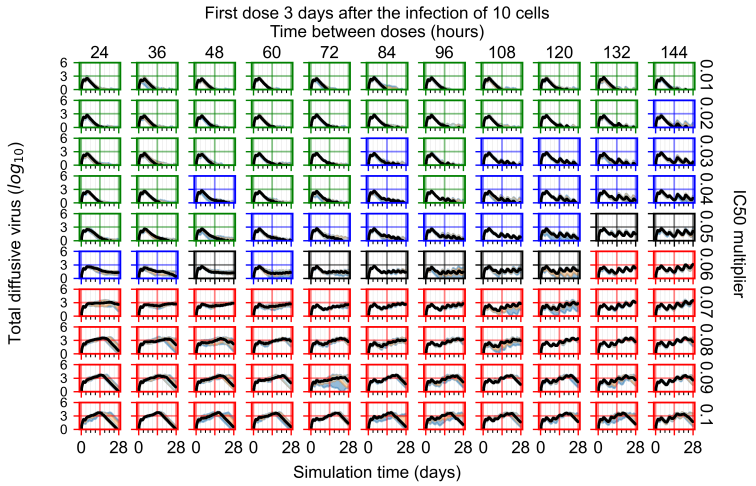


Figure A.34: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

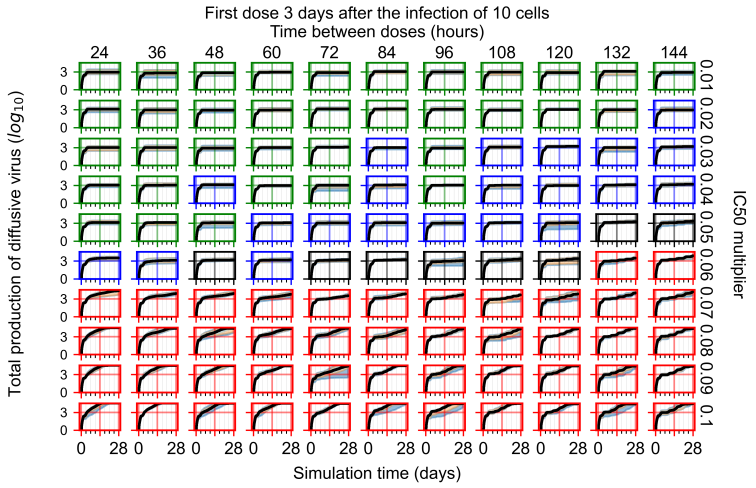


Figure A.35: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

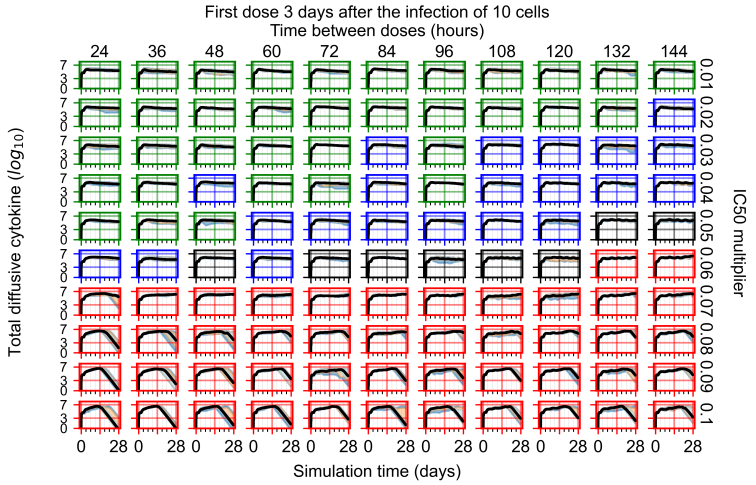


Figure A.36: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

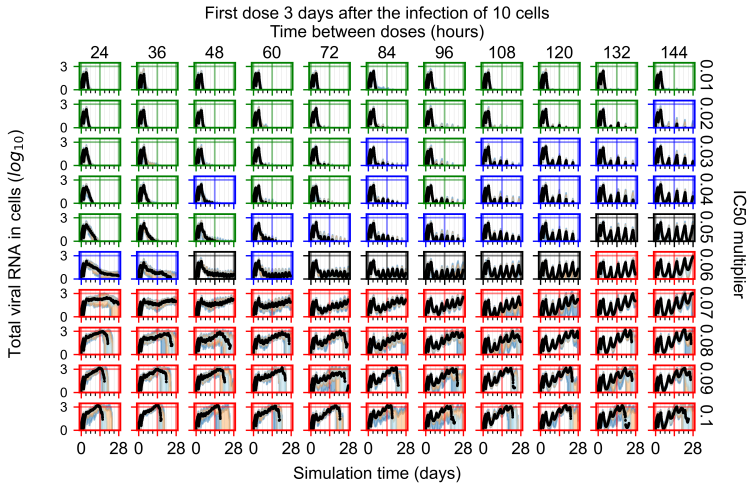


Figure A.37: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

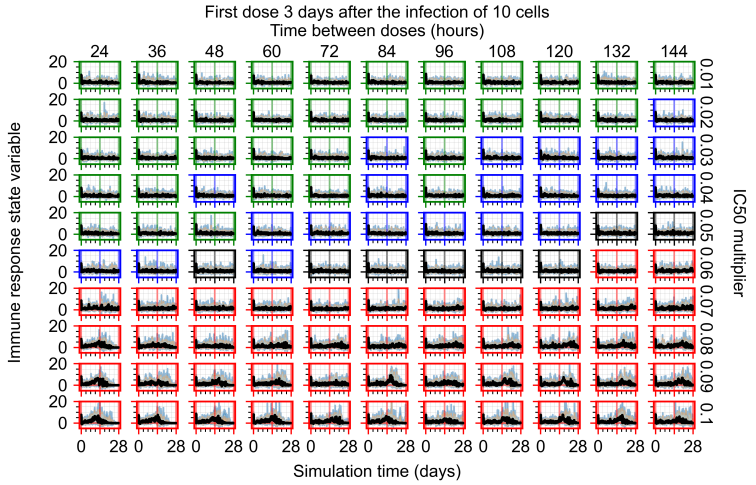


Figure A.38: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.2 Homogeneous metabolism, halved GS–443902 half-life

A.6.2.1 Treatment initiation with infection of ten epithelial cells

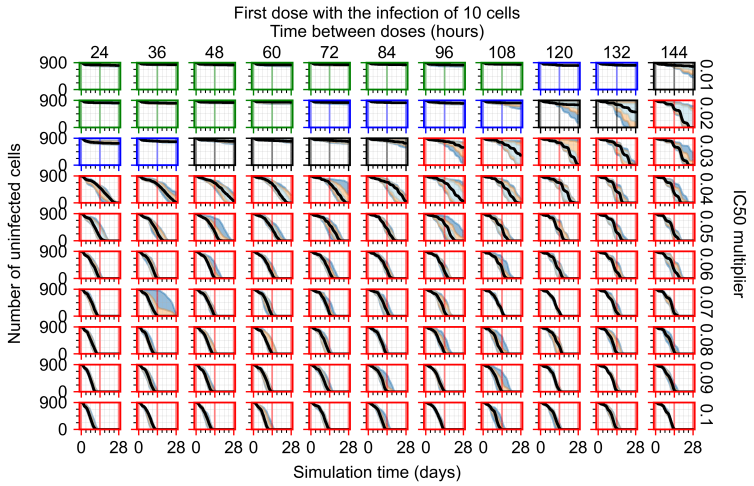


Figure A.39: Uninfected population.

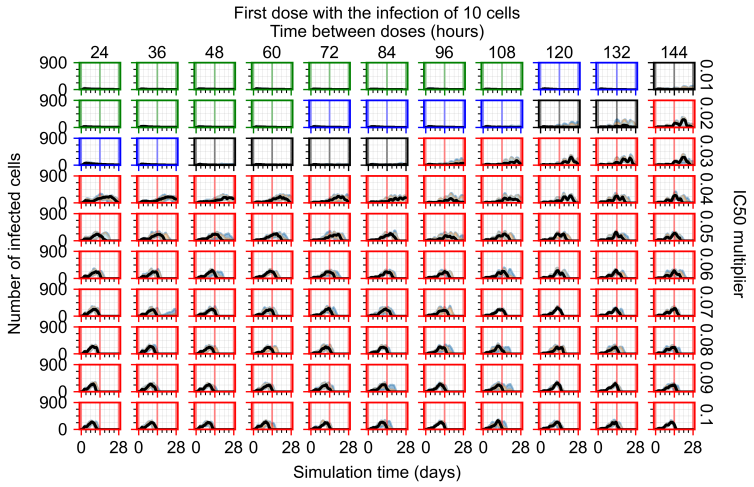


Figure A.40: Infected (eclipse phase) population.

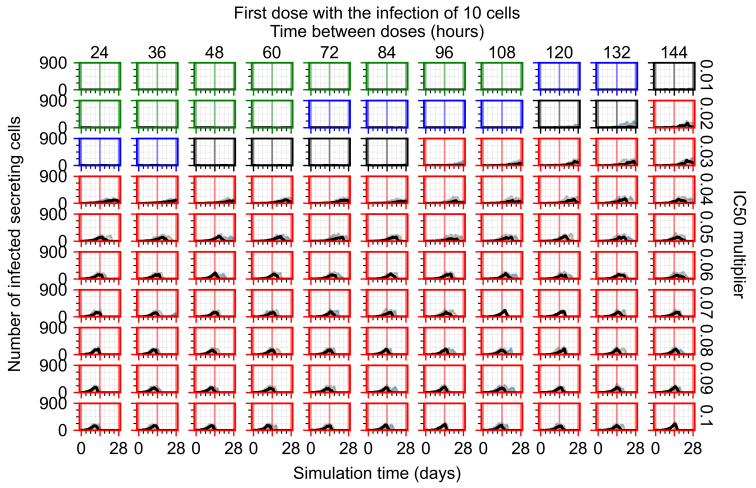


Figure A.41: Infected (secreting extracellular virus) population.

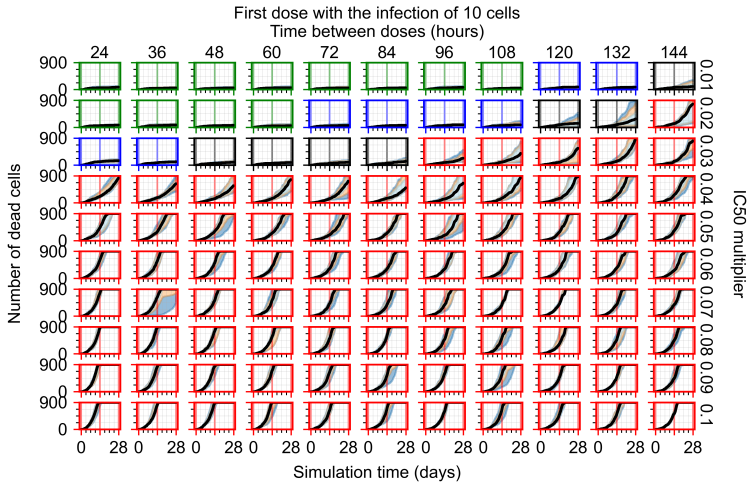


Figure A.42: Dead population.

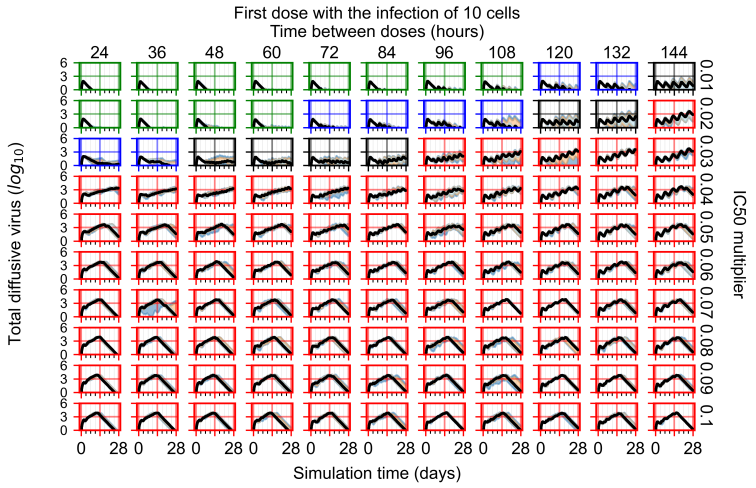


Figure A.43: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

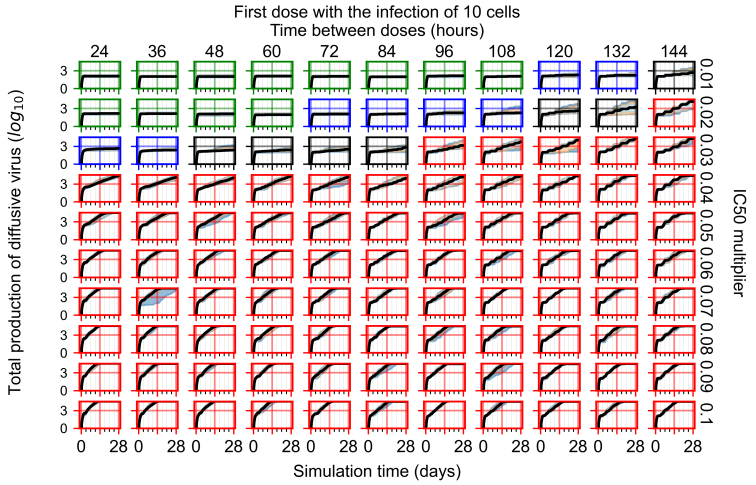


Figure A.44: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

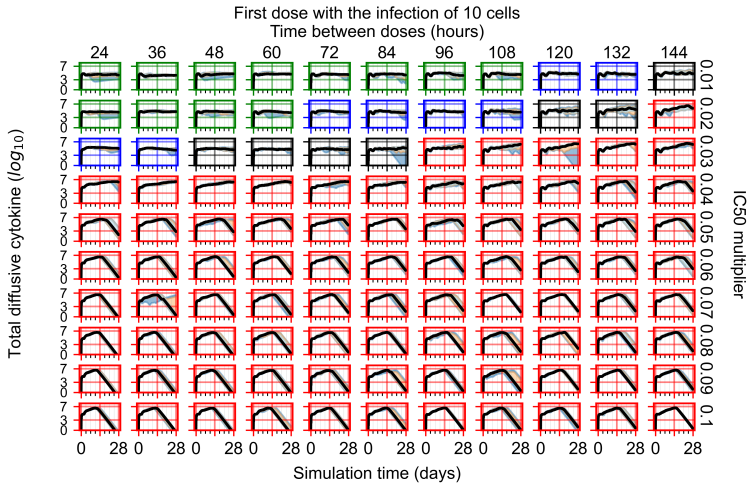


Figure A.45: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

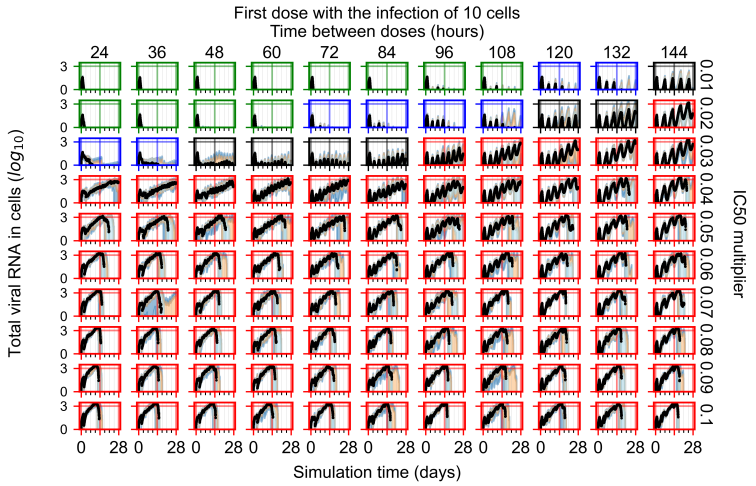


Figure A.46: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

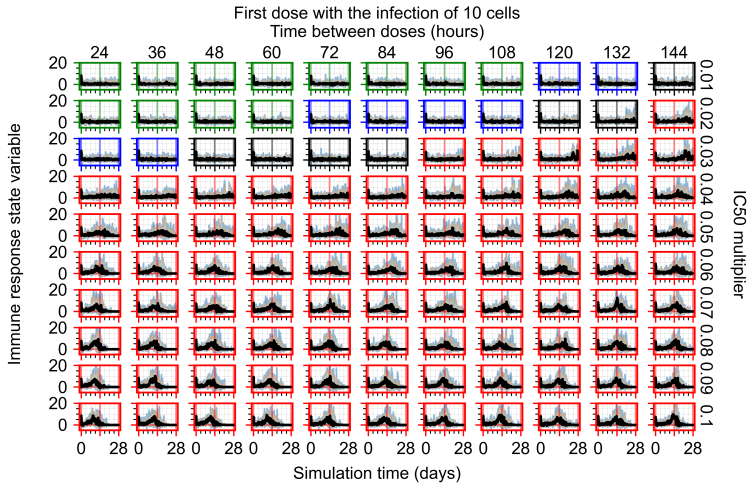


Figure A.47: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.2.2 Treatment initiation one day post infection of ten epithelial cells

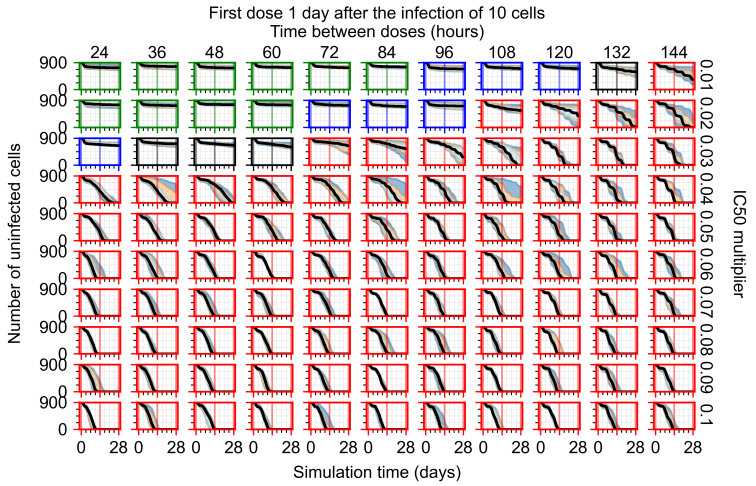


Figure A.48: Uninfected population

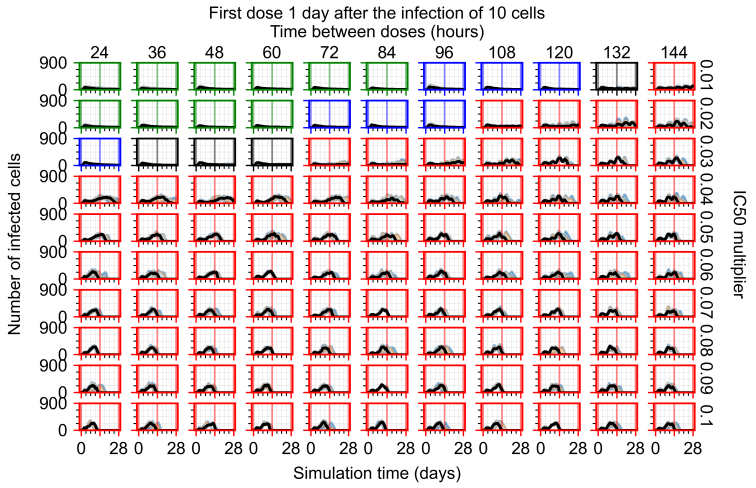


Figure A.49: Infected (eclipse phase) population.

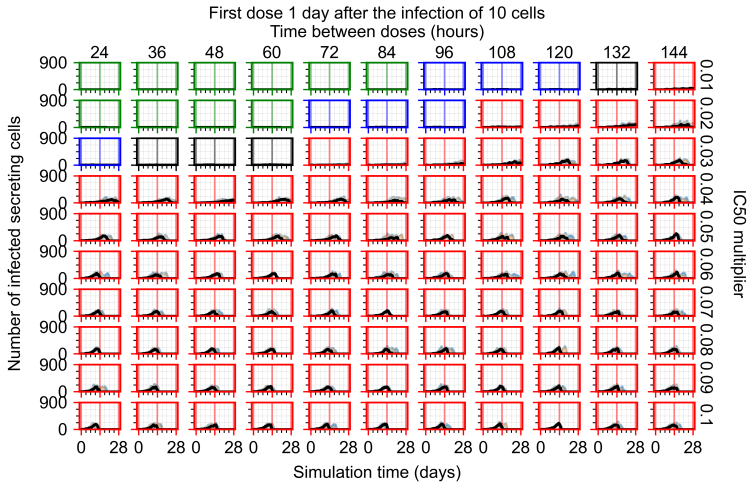


Figure A.50: Infected (secreting extracellular virus).

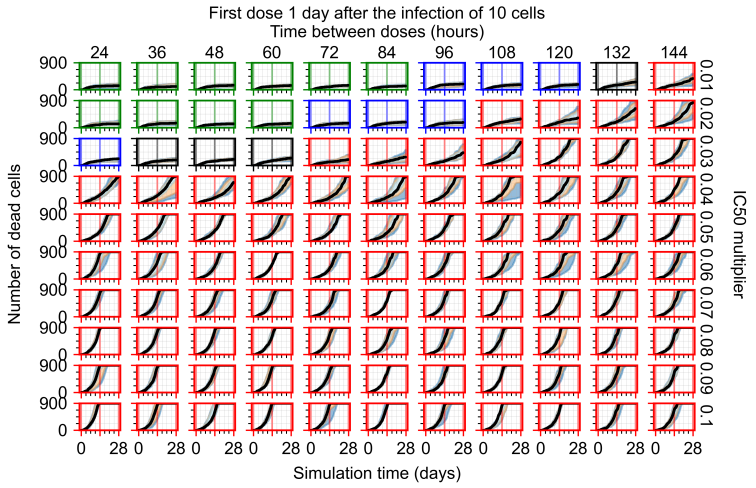


Figure A.51: Dead population.

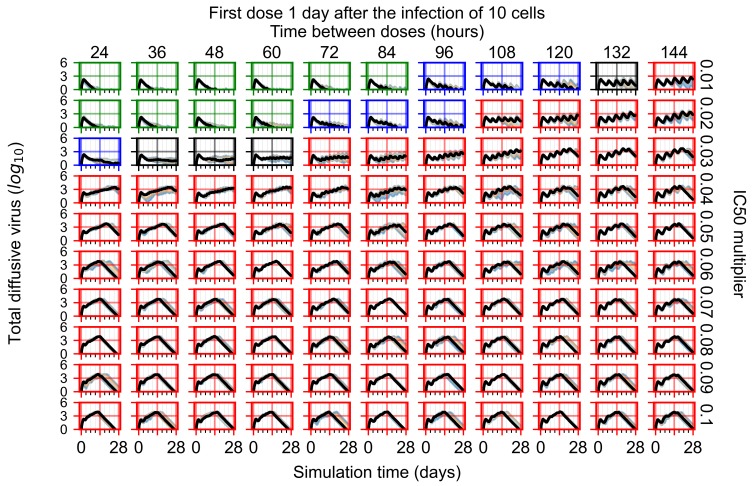


Figure A.52: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

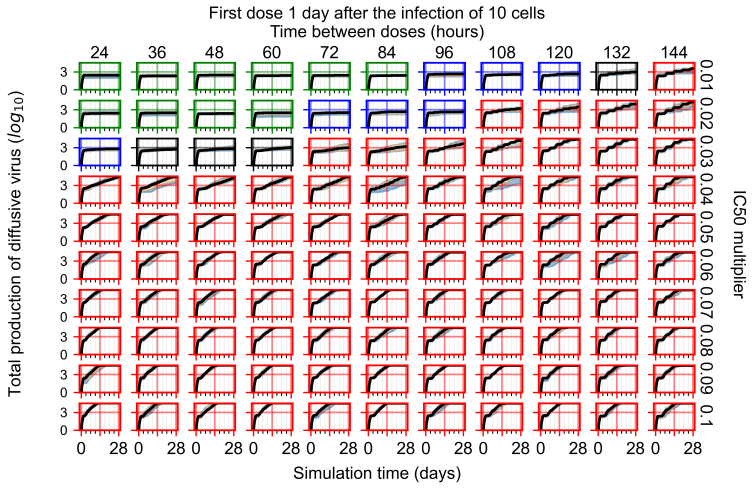


Figure A.53: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

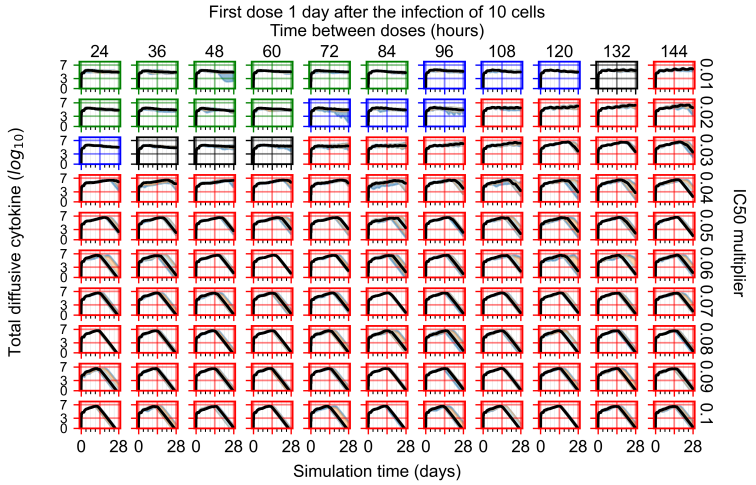


Figure A.54: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

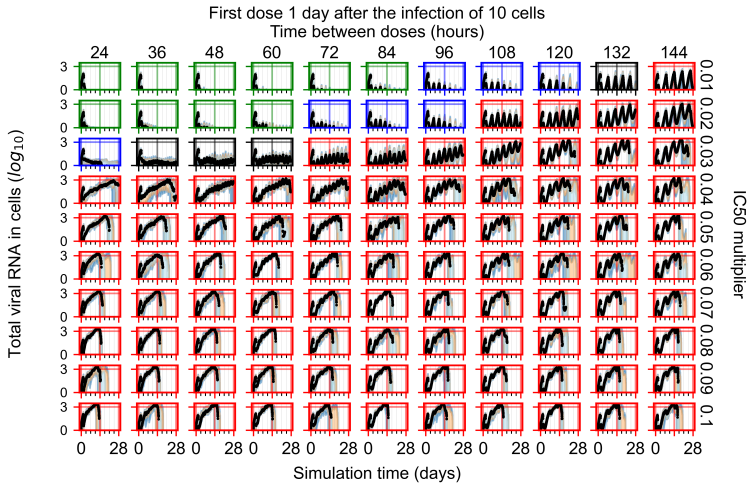


Figure A.55: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

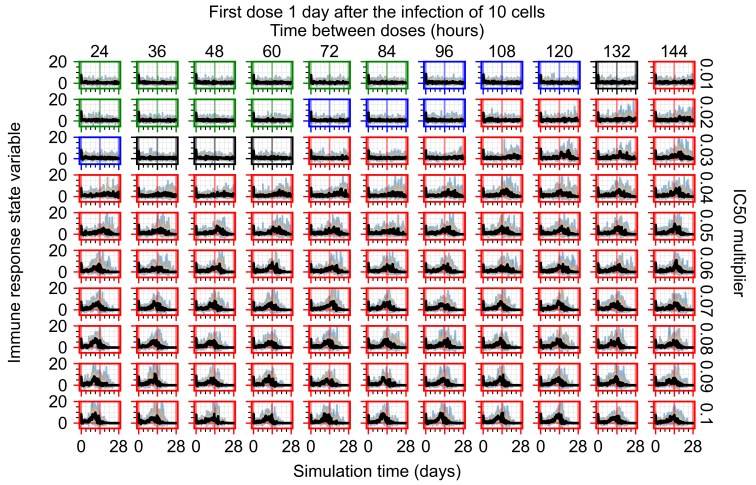


Figure A.56: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.2.3 Treatment initiation three days post infection of ten epithelial cells

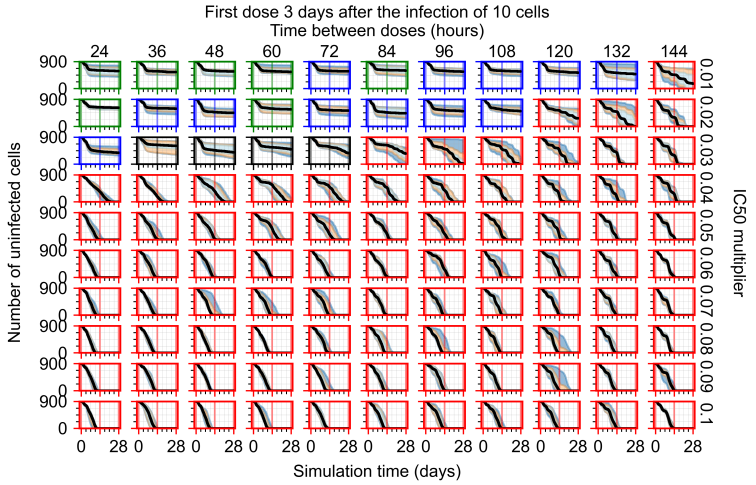


Figure A.57: Uninfected population.

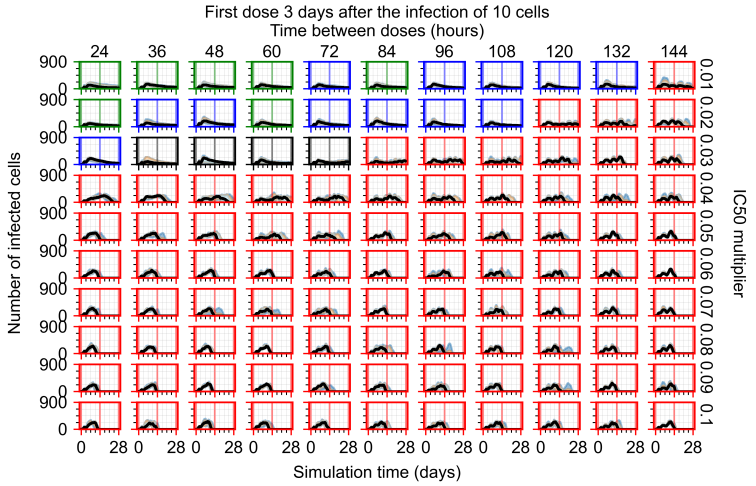


Figure A.58: Infected (eclipse phase) population.

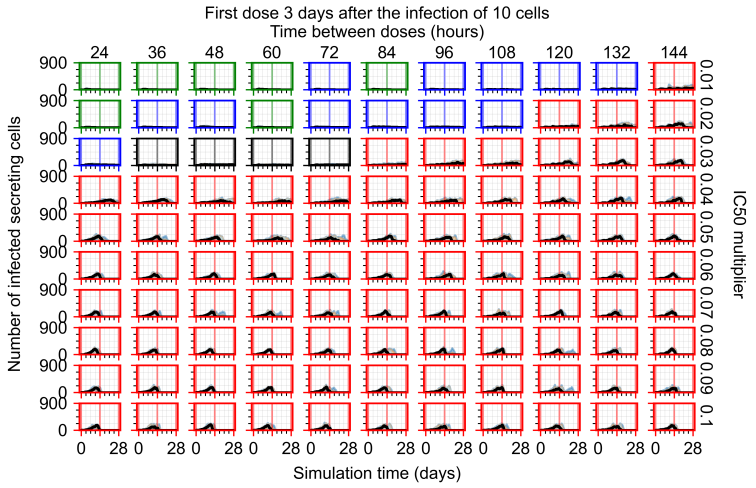


Figure A.59: Infected (secreting extracellular virus) population.

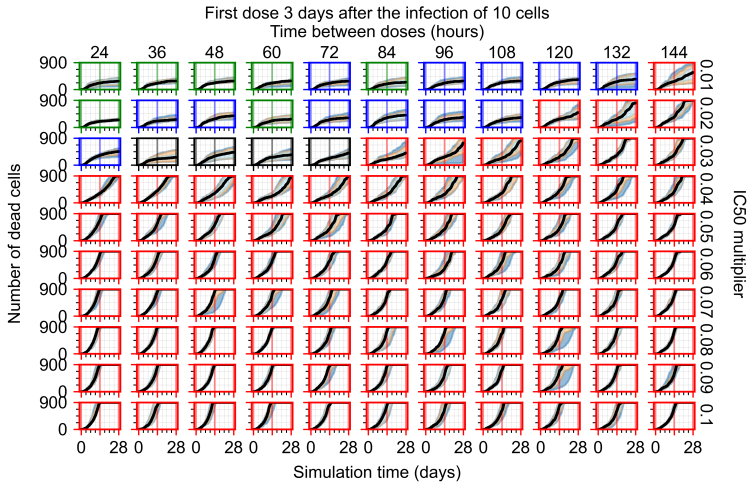


Figure A.60: Dead population.

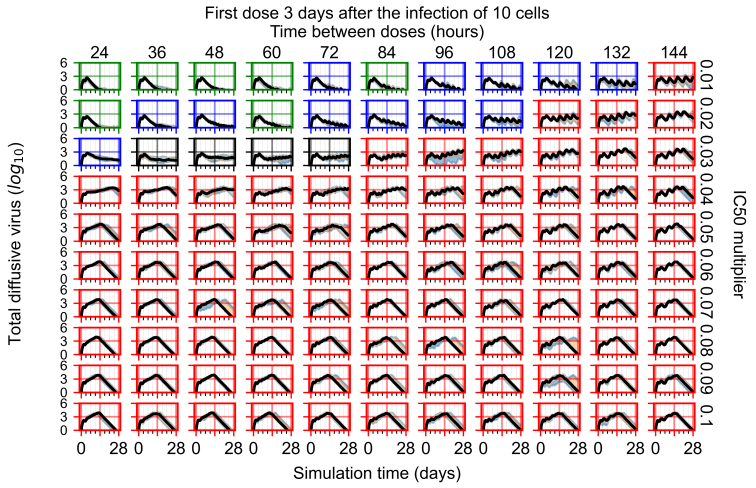


Figure A.61: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

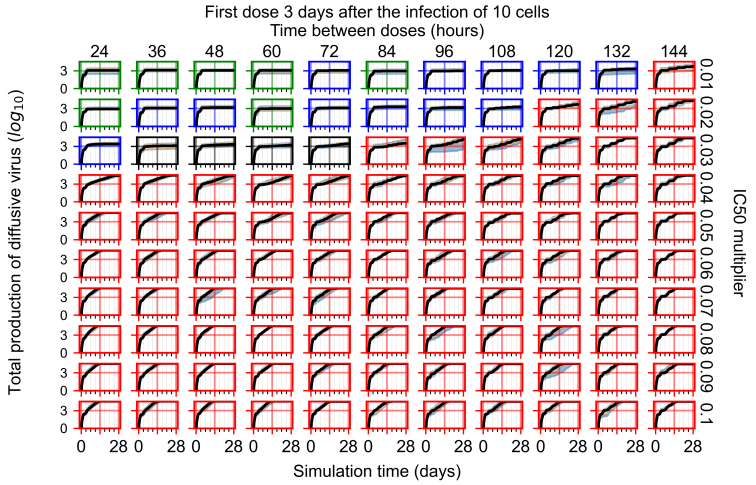


Figure A.62: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

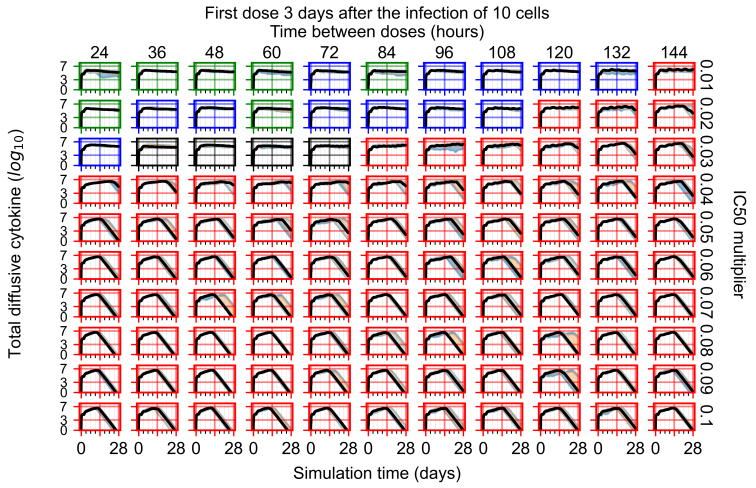


Figure A.63: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

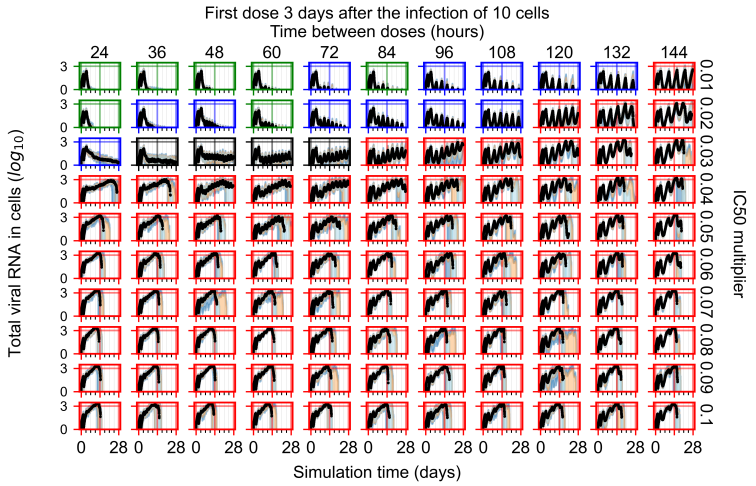


Figure A.64: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

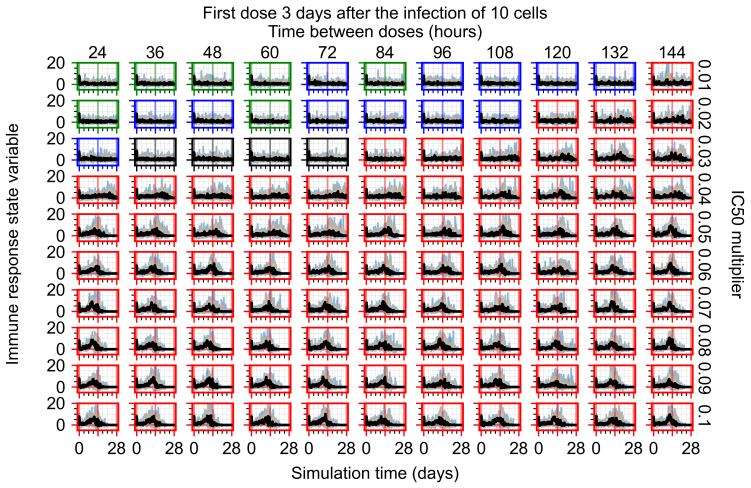


Figure A.65: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.3 Homogeneous metabolism, GS–443902 half-life reduced by 75%

A.6.3.1 Treatment initiation with infection of ten epithelial cells

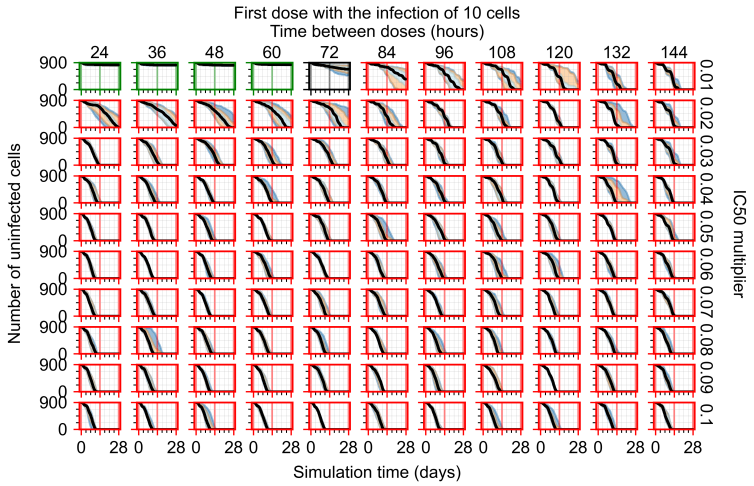


Figure A.66: Uninfected population.

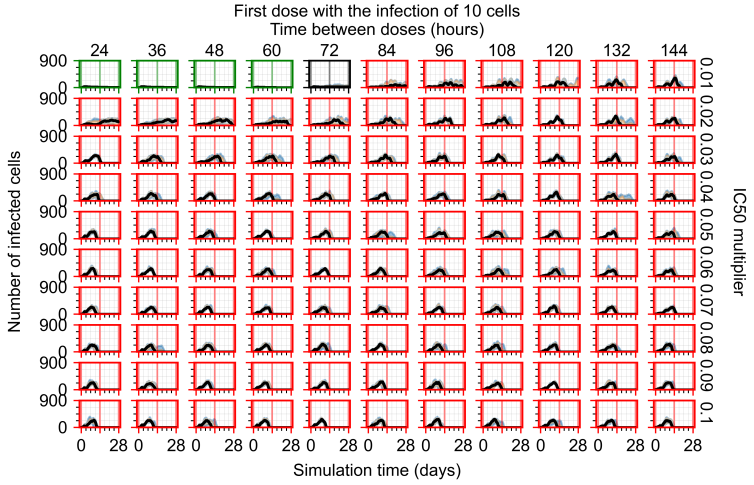


Figure A.67: Infected (eclipse phase) population.

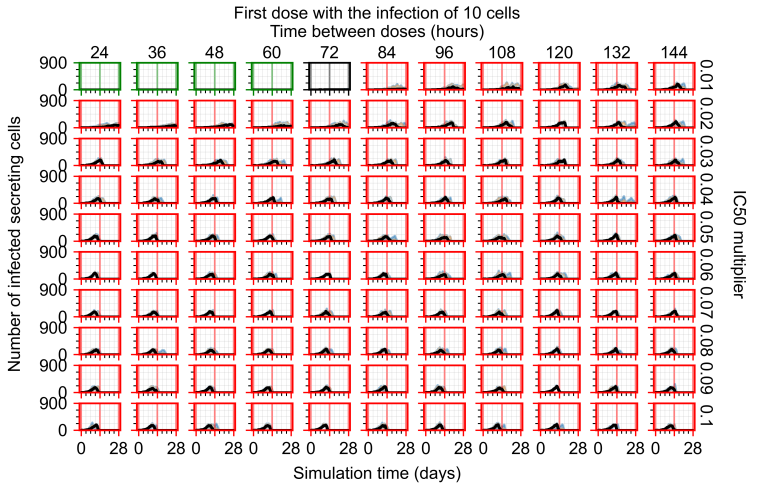


Figure A.68: Infected (secreting extracellular virus) population.

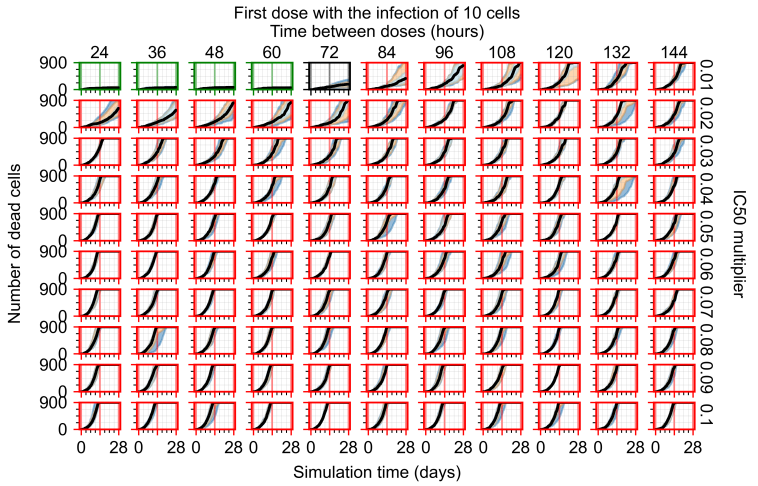


Figure A.69: Dead population.

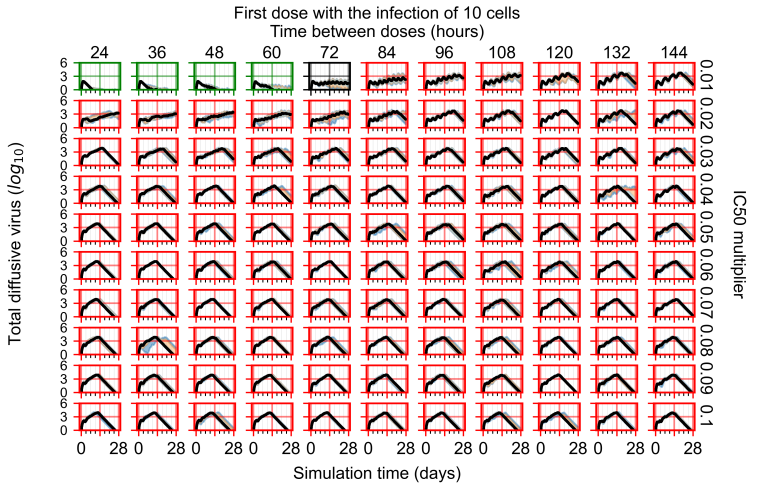


Figure A.70: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

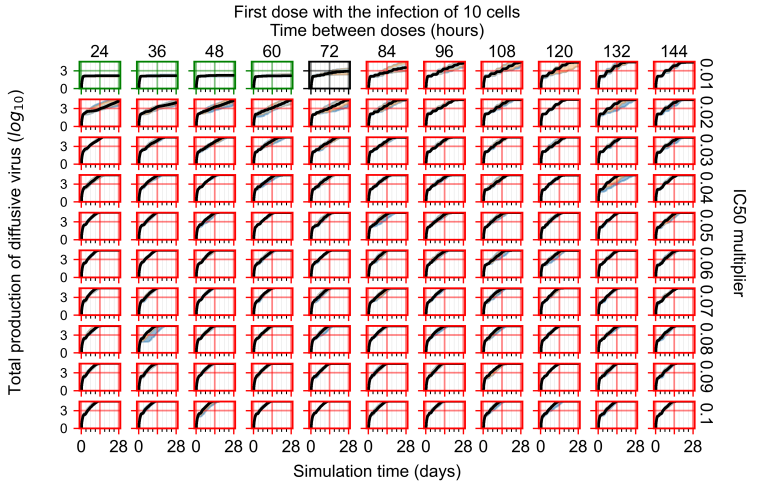


Figure A.71: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

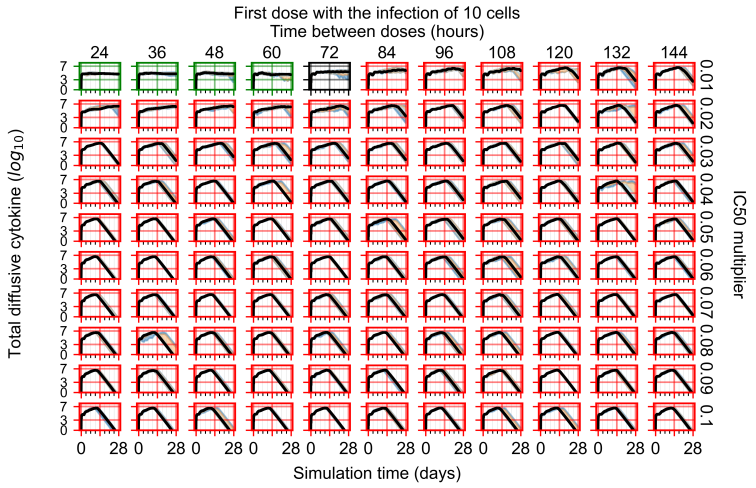


Figure A.72: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

Figure A.73: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

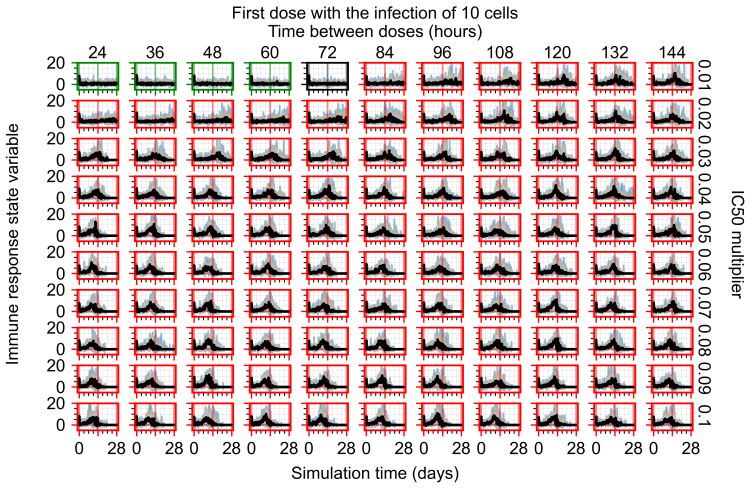


Figure A.74: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.4 Heterogeneous metabolism, regular GS–443902 half-life

A.6.4.1 Treatment initiation with infection of ten epithelial cells

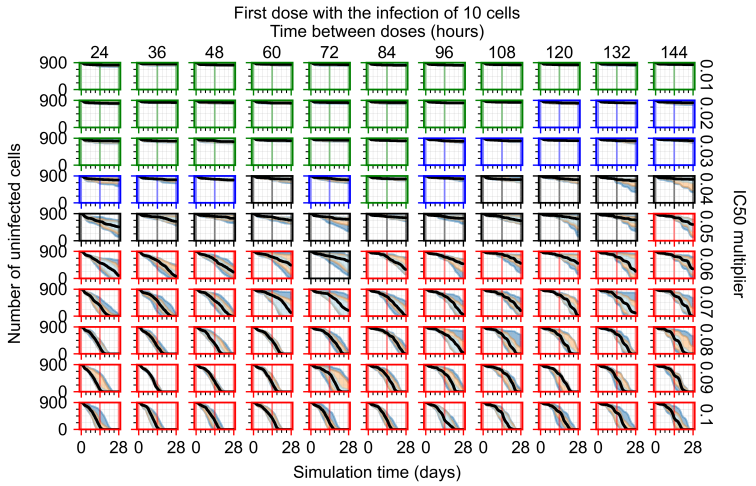


Figure A.75: Uninfected population.

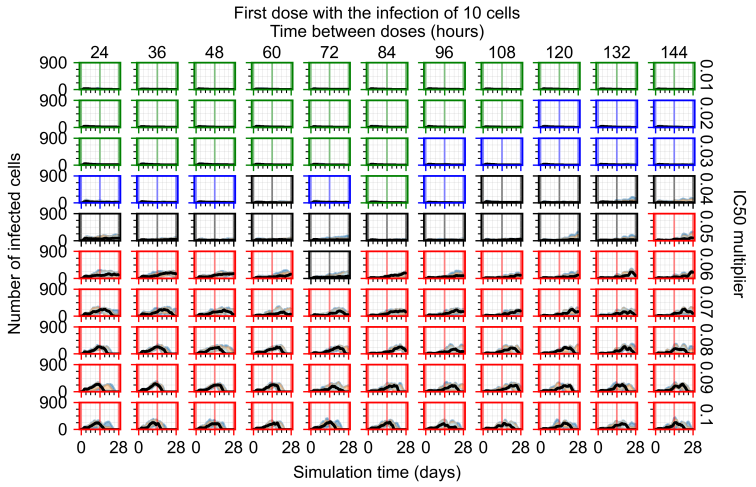


Figure A.76: Infected (eclipse phase) population.

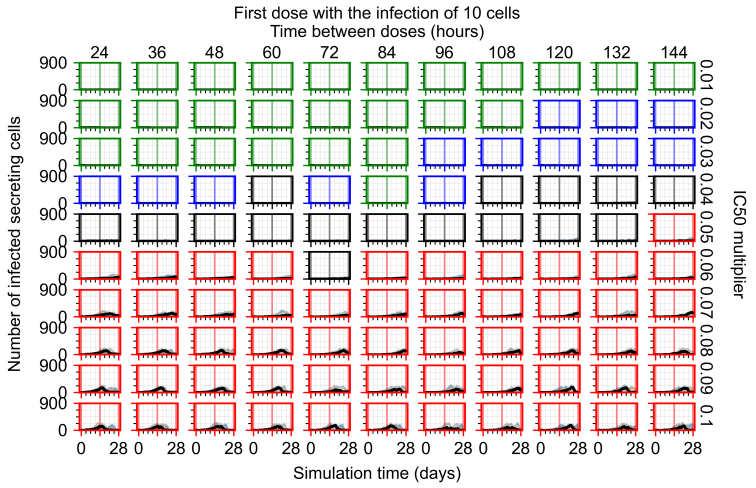


Figure A.77: Infected (secreting extracellular virus) population.

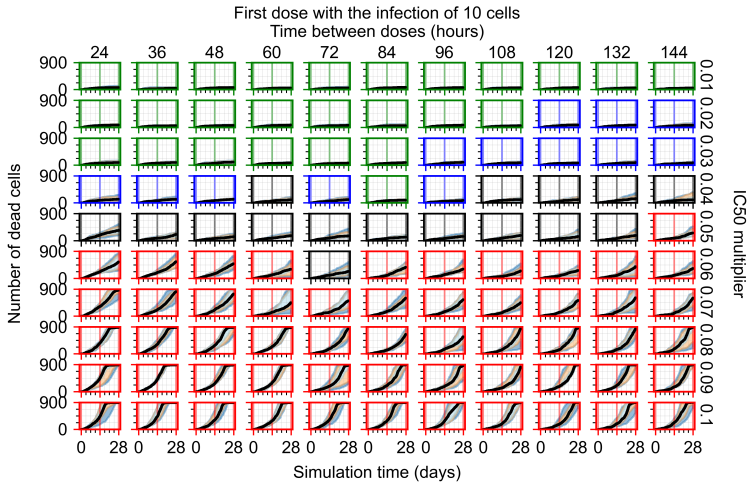


Figure A.78: Dead population.

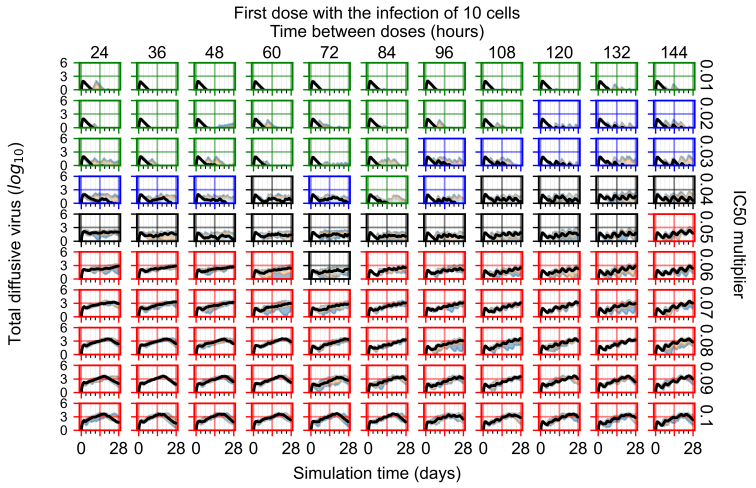


Figure A.79: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

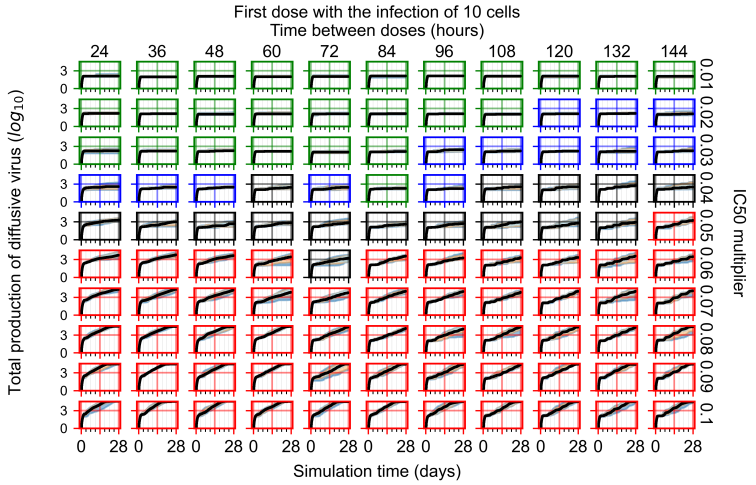


Figure A.80: Total diffusive virus produced (AUC) for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

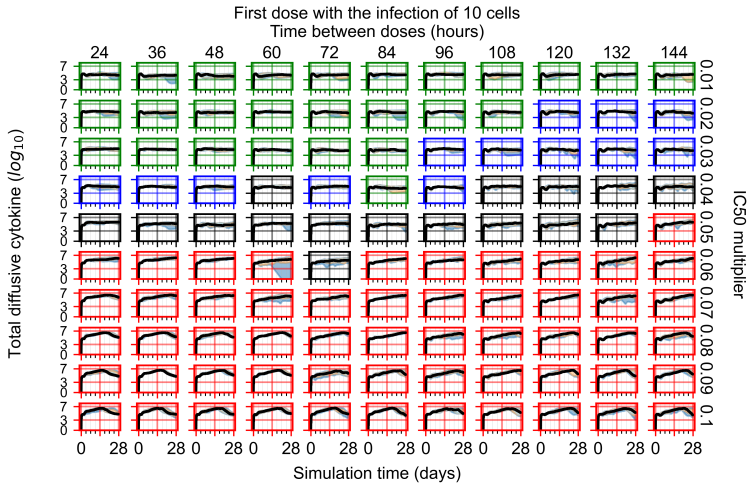


Figure A.81: Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

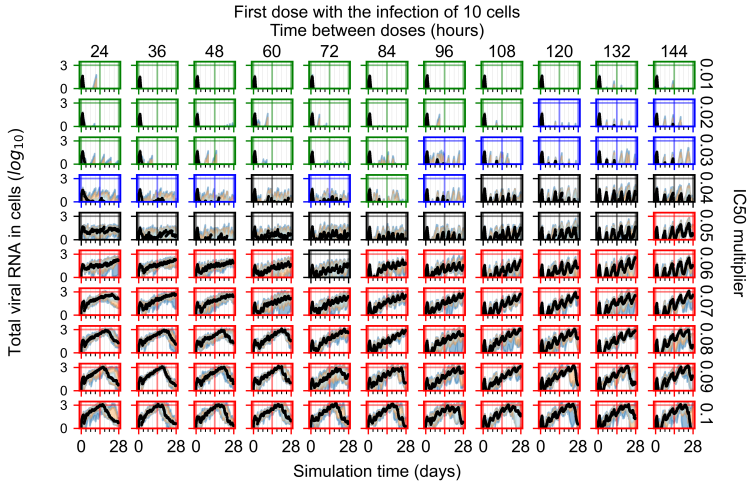


Figure A.82: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

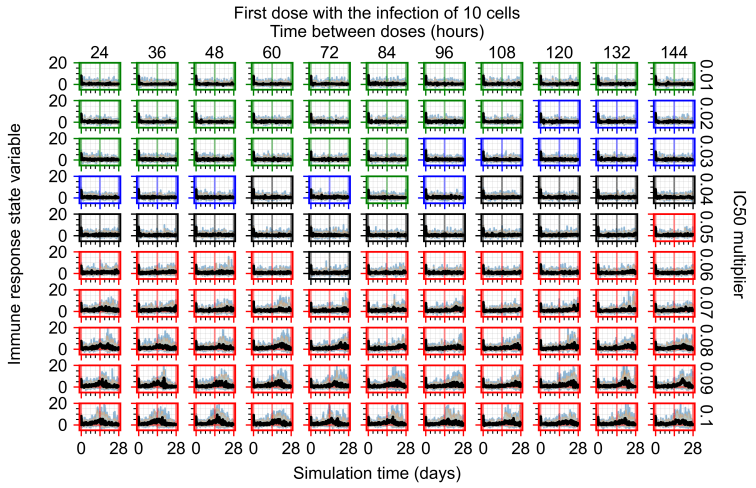


Figure A.83: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.4.2 Treatment initiation twelve hours post the infection of ten epithelial cells

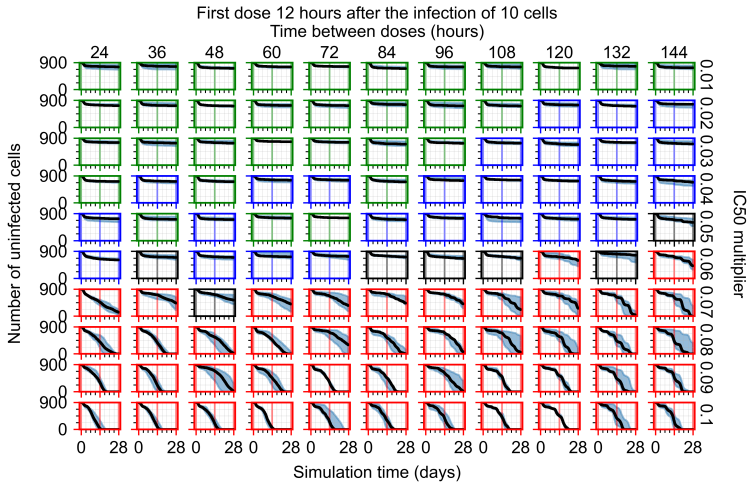


Figure A.84: Uninfected population.

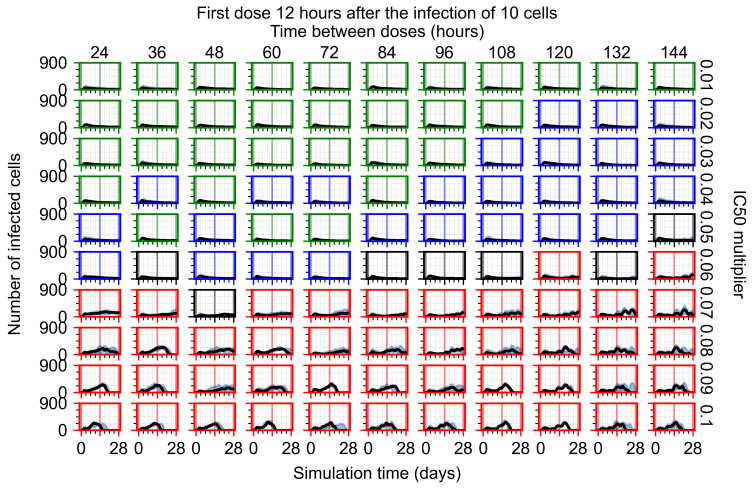


Figure A.85: Infected (eclipse phase) population.

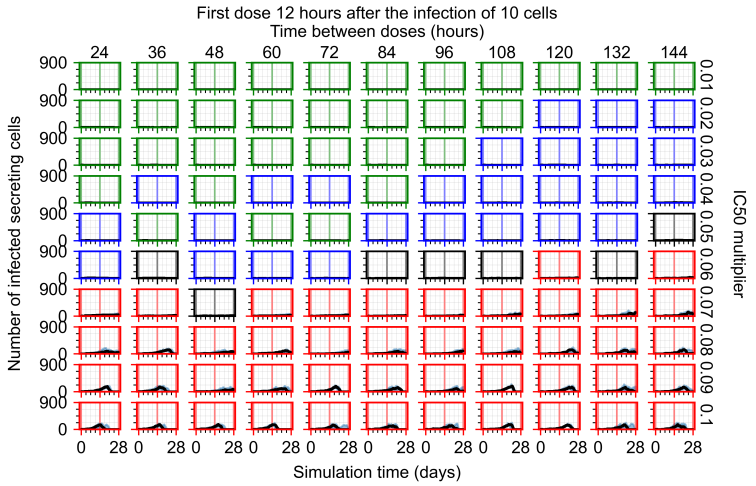


Figure A.86: Infected (secreting extracellular virus) population.

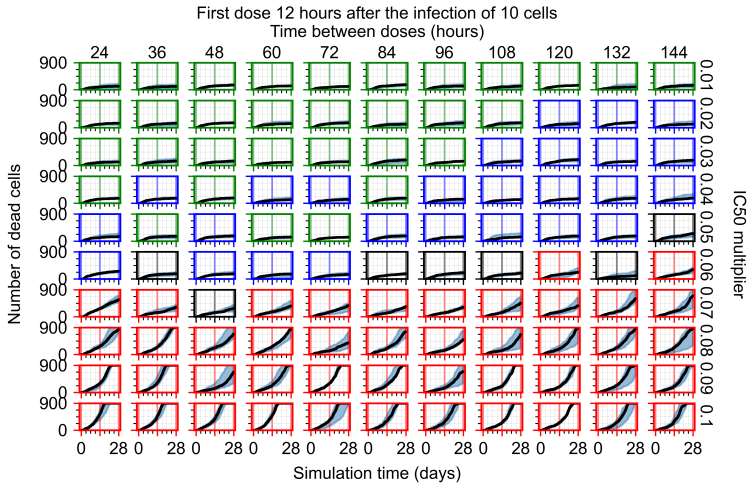


Figure A.87: Dead population.

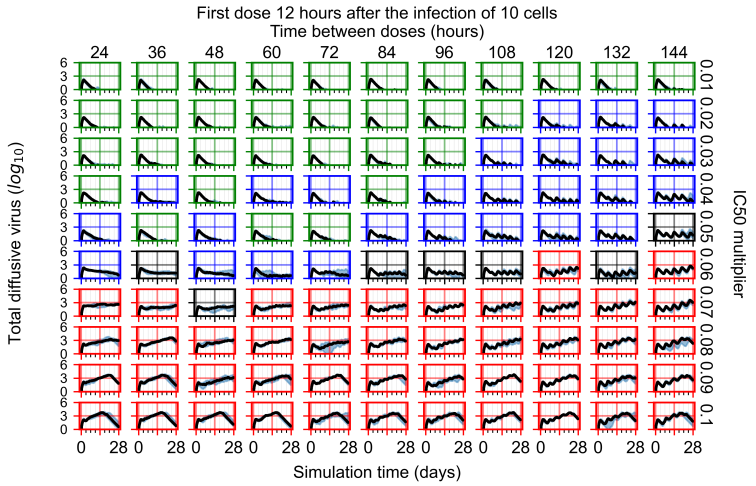


Figure A.88: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

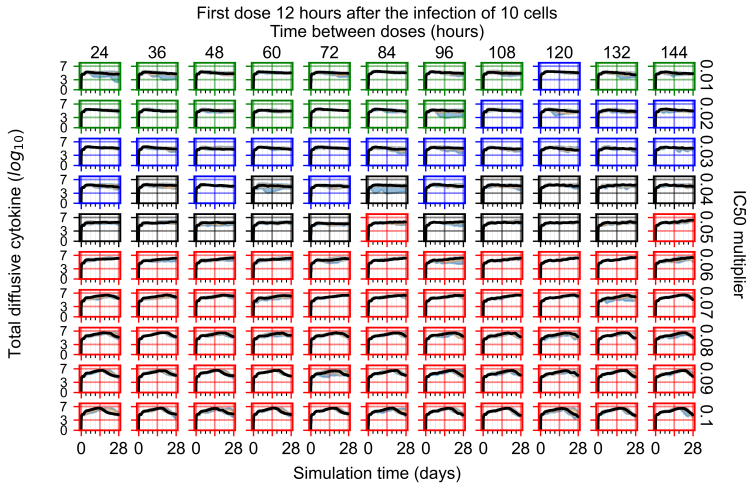


Figure A.89: Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

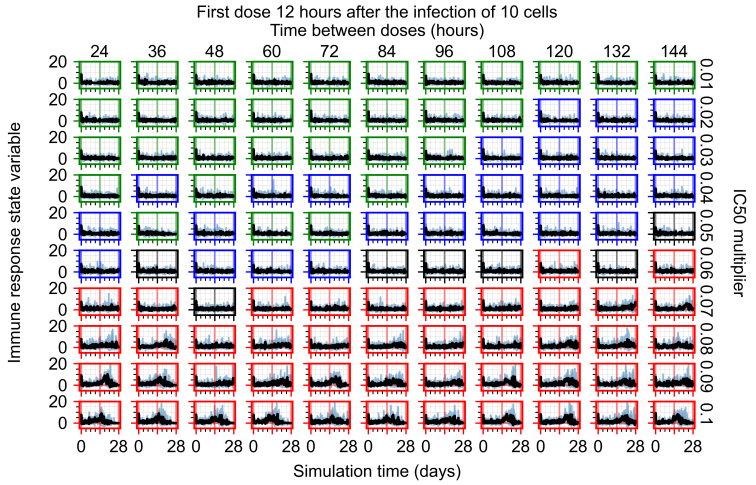


Figure A.90: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.4.3 Treatment initiation one day post the infection of ten epithelial cells

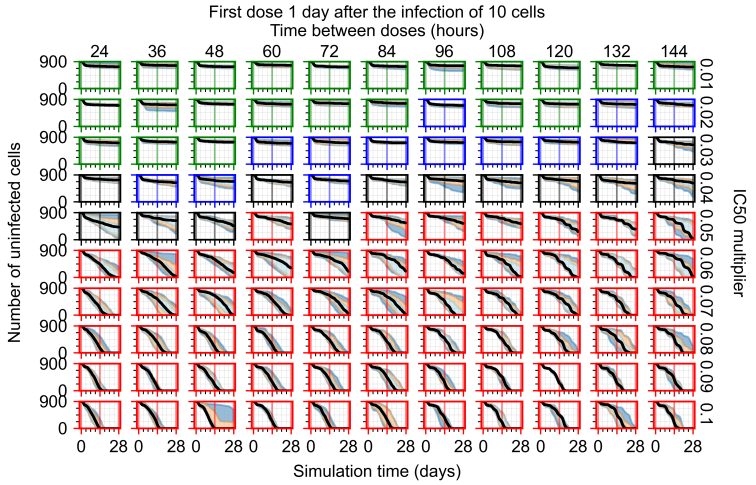


Figure A.91: Uninfected population.

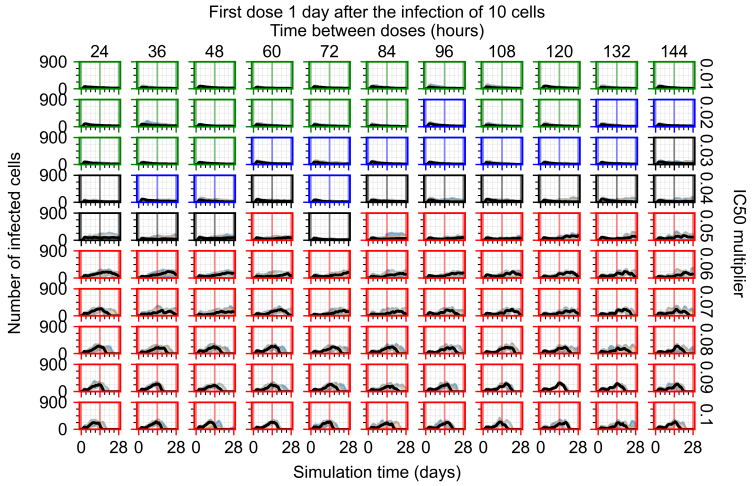


Figure A.92: Infected (eclipse phase) population.

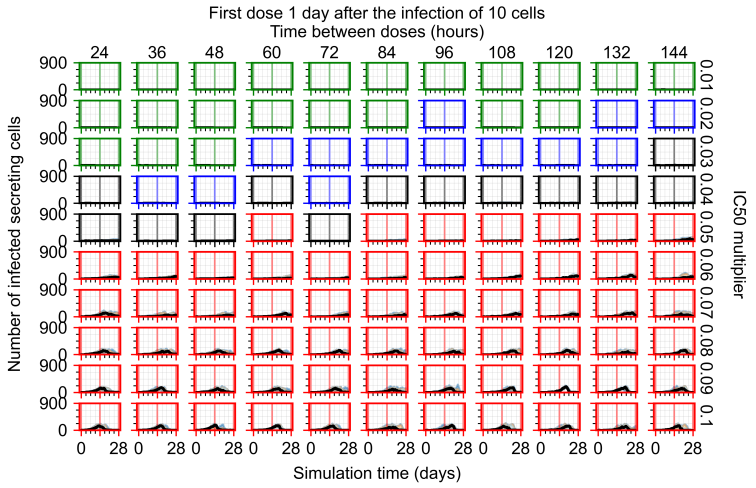


Figure A.93: Infected (secreting extracellular virus) population.

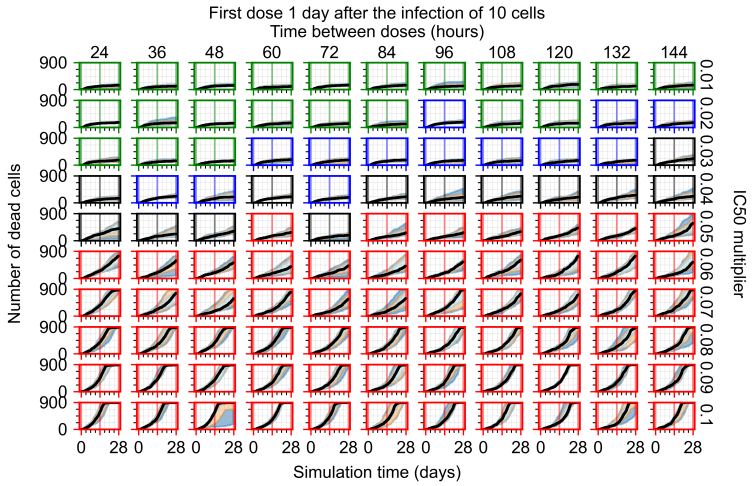


Figure A.94: Dead population.

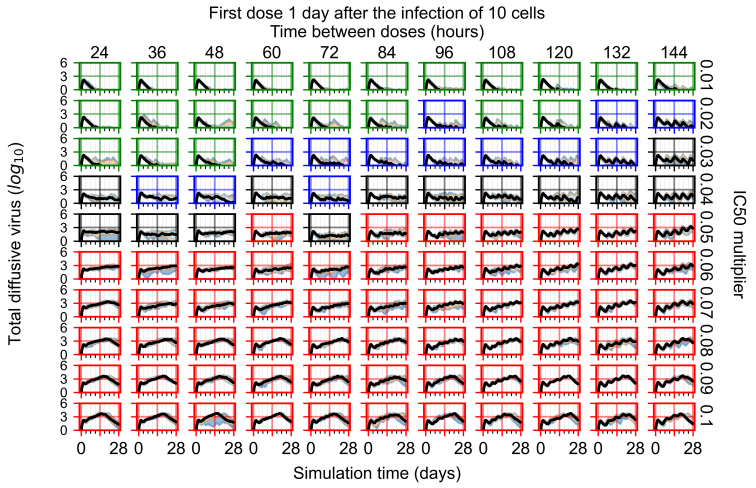


Figure A.95: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

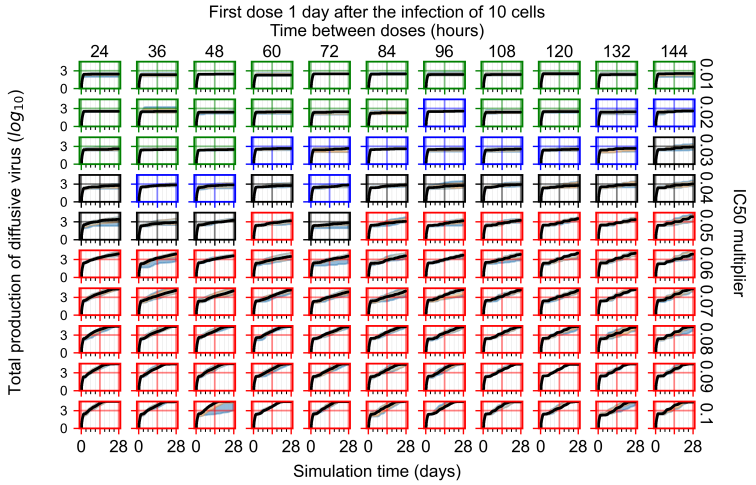


Figure A.96: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

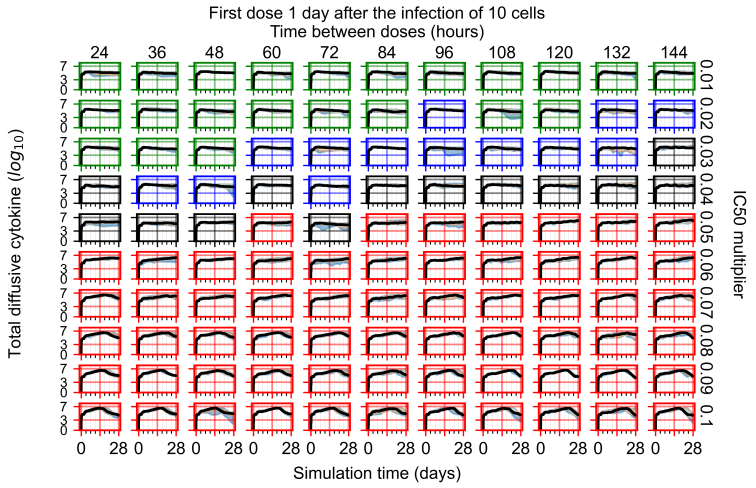


Figure A.97: Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

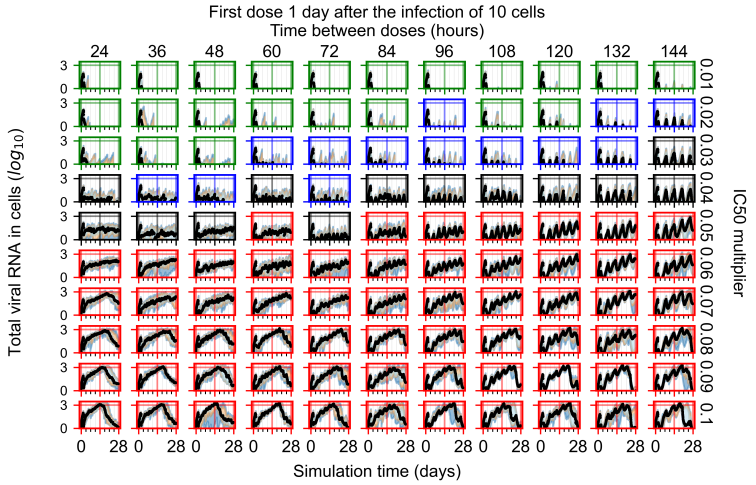


Figure A.98: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

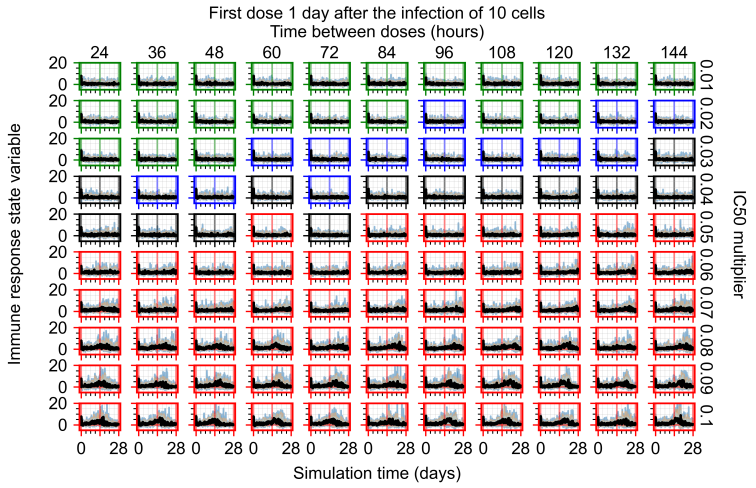


Figure A.99: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.4.4 Treatment initiation three days post the infection of ten epithelial cells

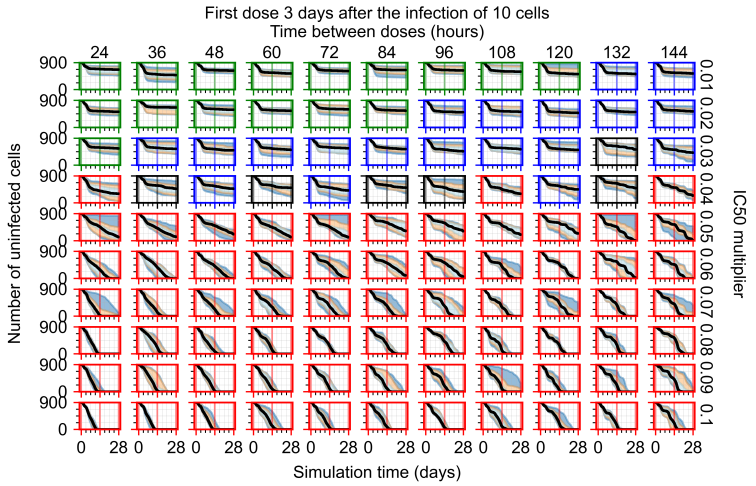


Figure A.100: Uninfected population.

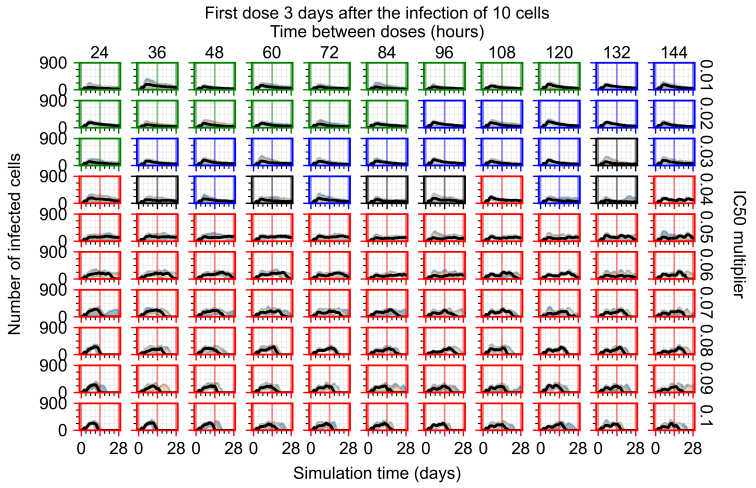


Figure A.101: Infected (eclipse phase) population.

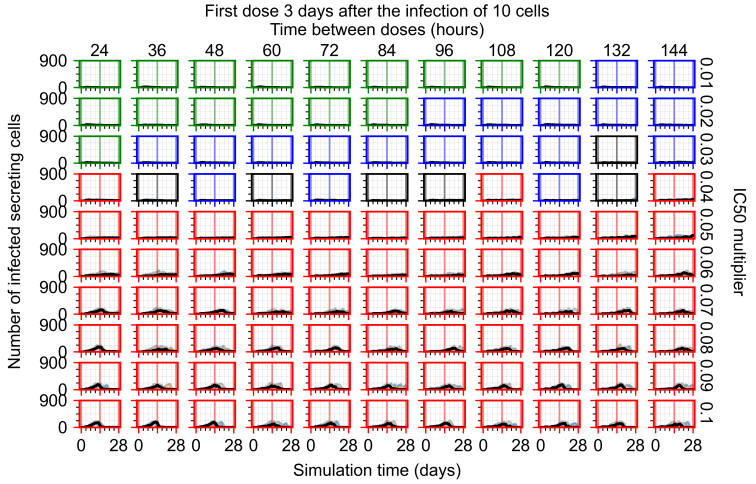


Figure A.102: Infected (secreting extracellular virus) population.

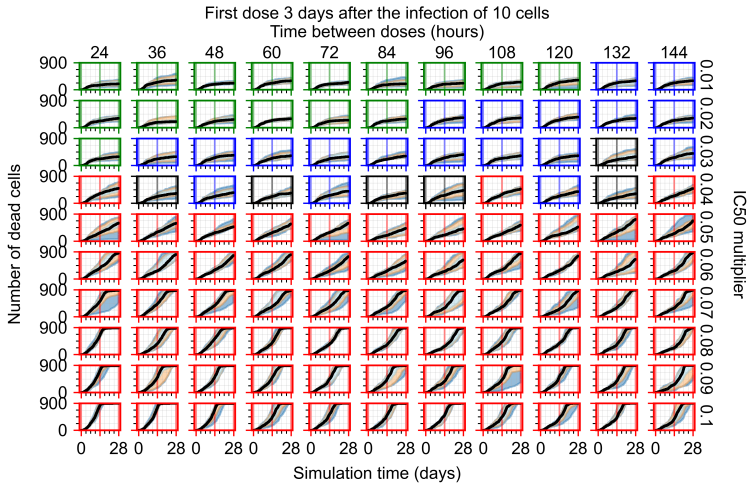


Figure A.103: Dead population.

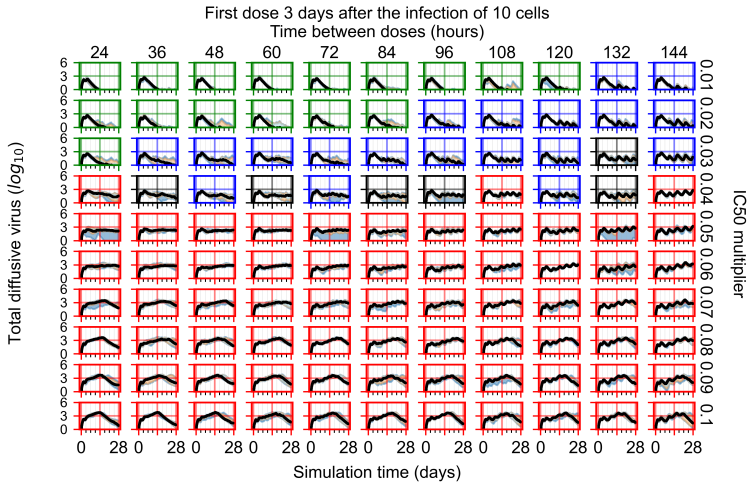


Figure A.104: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

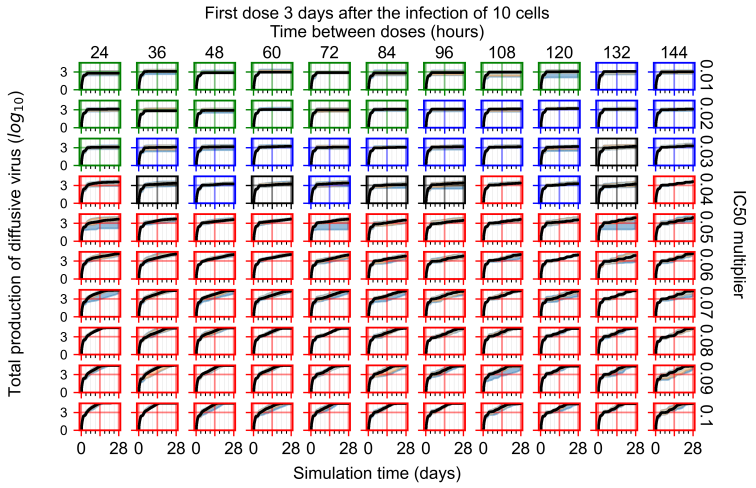


Figure A.105: Total diffusive virus produced (AUC), Y axis in log scale, exponent values as tick-marks.

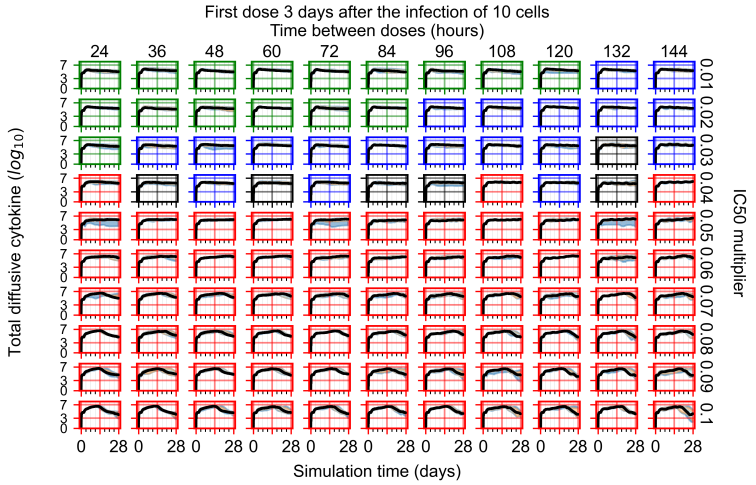


Figure A.106: Diffusive cytokine amount, Y axis in log scale, exponent values as tick-marks.

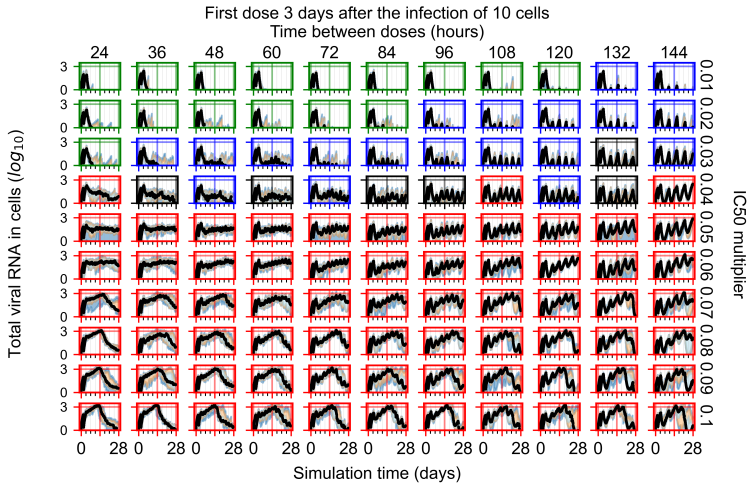


Figure A.107: Amount of viral RNA in infected cells (arbitrary units), Y axis in log scale, exponent values as tick-marks.

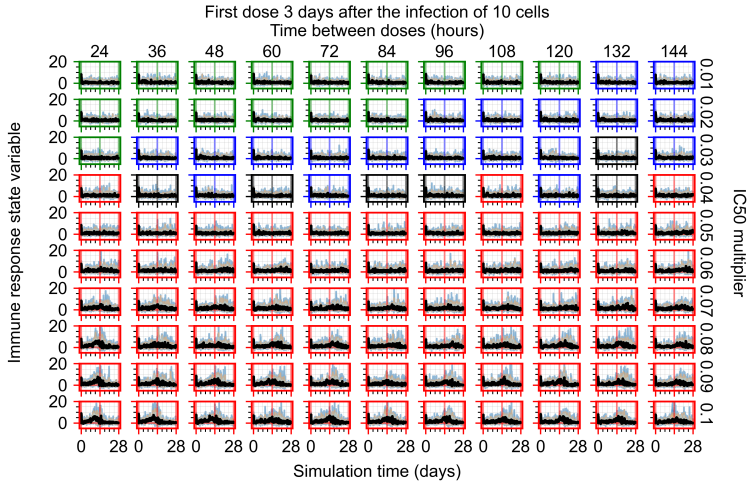


Figure A.108: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.5 Heterogeneous metabolism using other standard deviations

In these simulations treatment was initialized one day after the infection of 10 cells.

A.6.5.1 Standard deviation set to 0.1

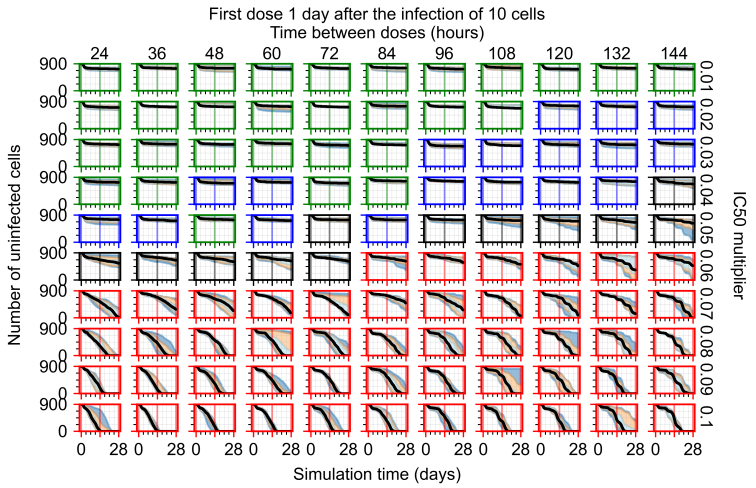


Figure A.109: Uninfected population.

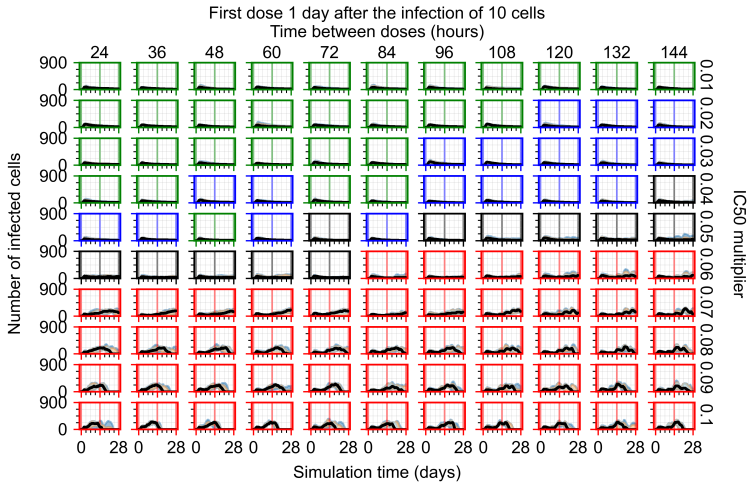


Figure A.110: Infected (eclipse phase) population.

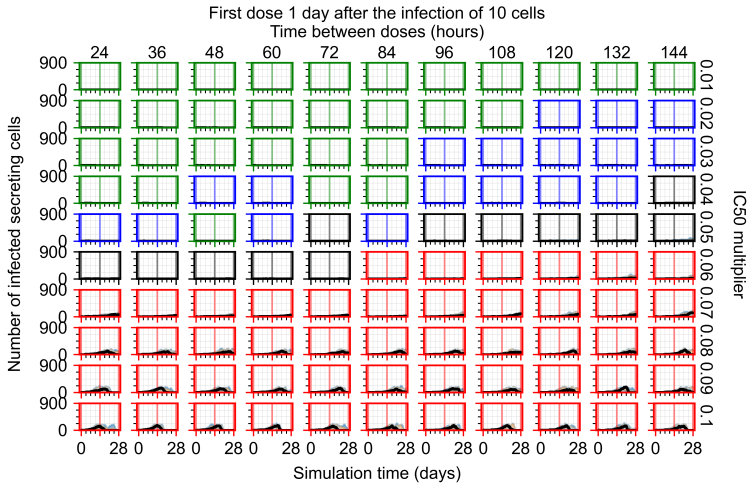


Figure A.111: Infected (secreting extracellular virus) population.

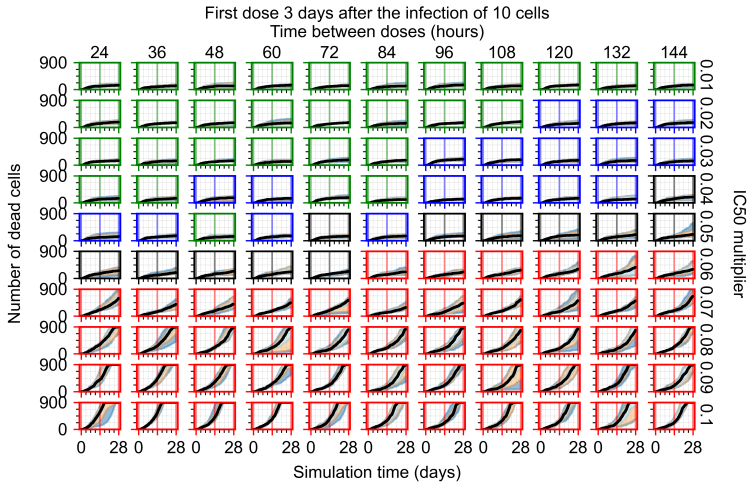


Figure A.112: Dead population.

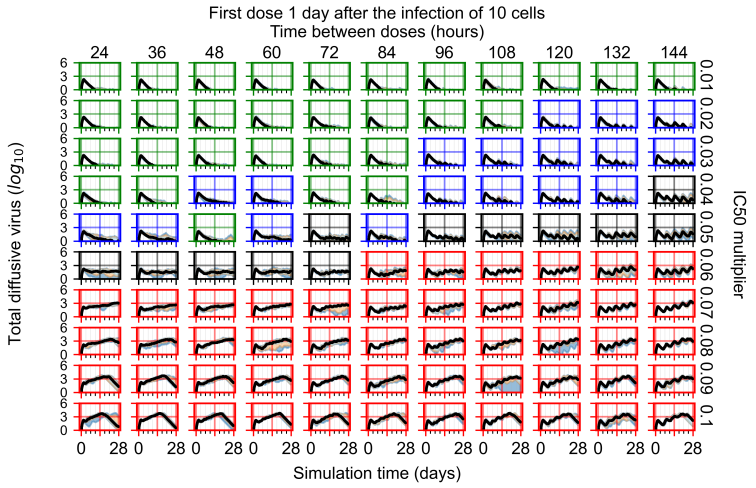


Figure A.113: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

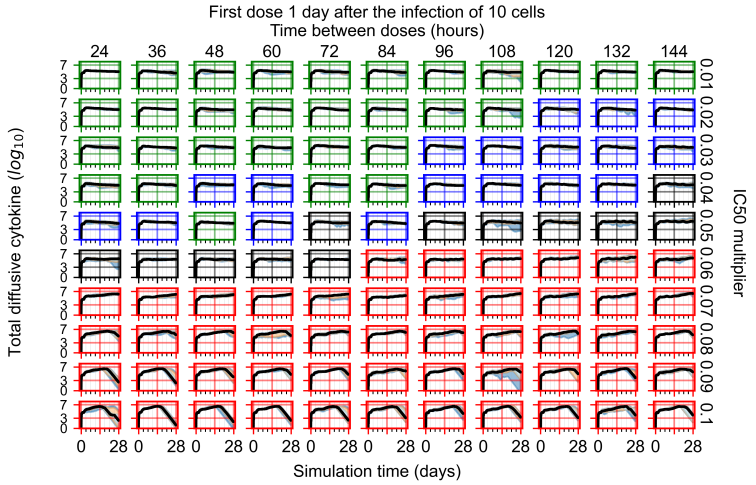


Figure A.114: Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

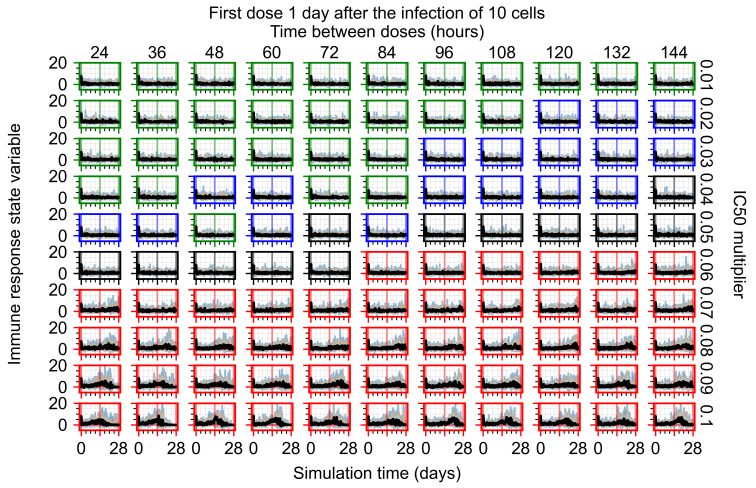


Figure A.115: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.5.2 Standard deviation set to 0.5

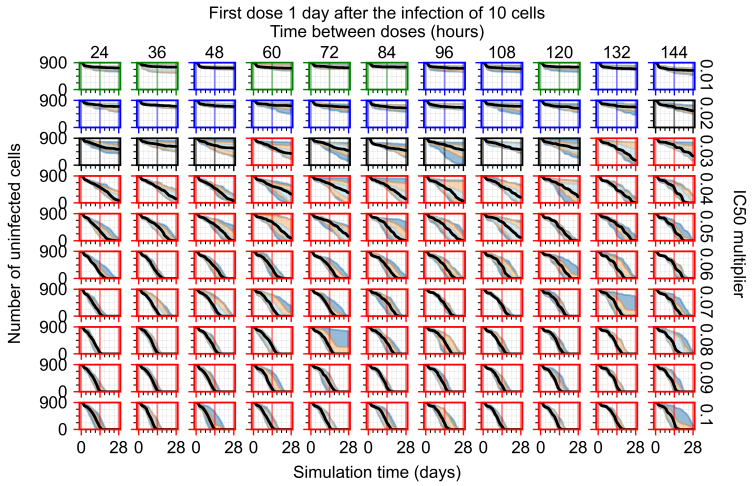


Figure A.116: Uninfected population.

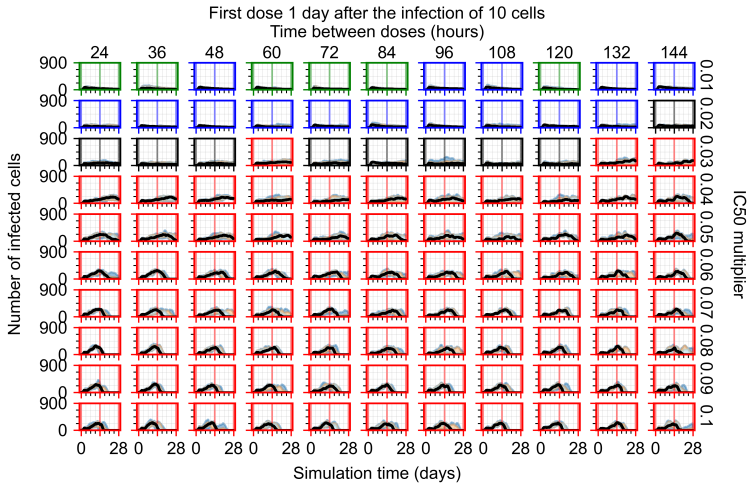


Figure A.117: Infected (eclipse phase) population.

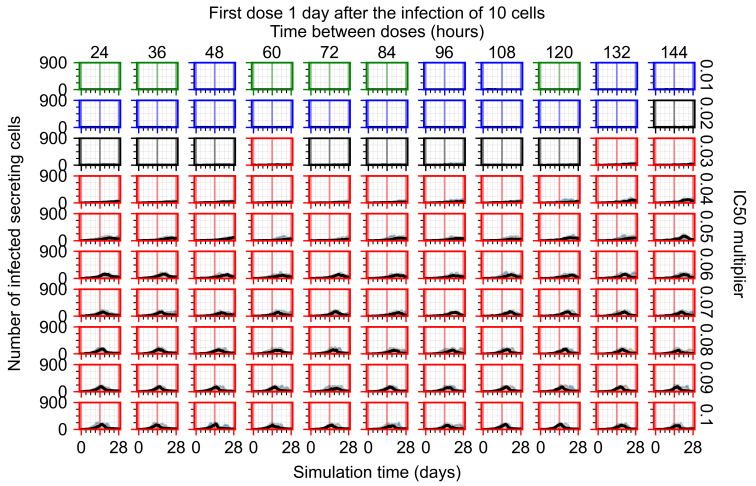


Figure A.118: Infected (secreting extracellular virus) population.

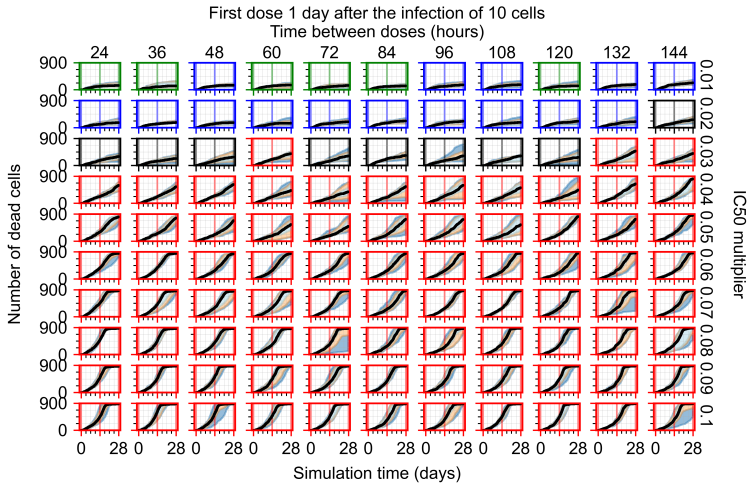


Figure A.119: Dead population.

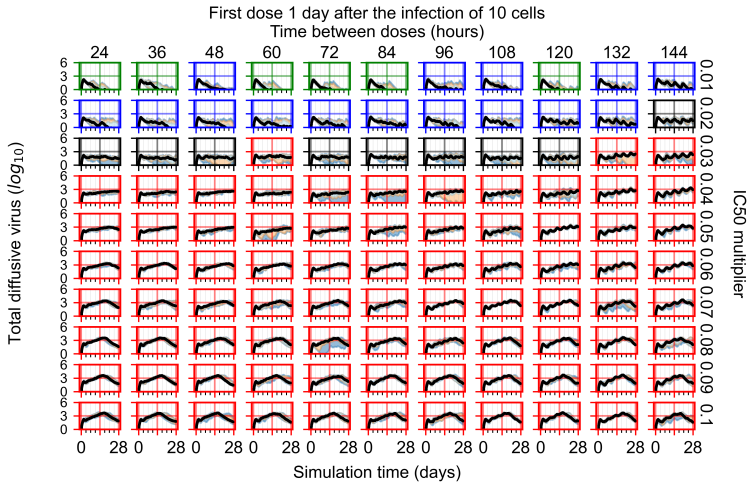


Figure A.120: Extracellular diffusive virus quantities (arbitrary units), Y axis in log scale, exponent values as tick-marks.

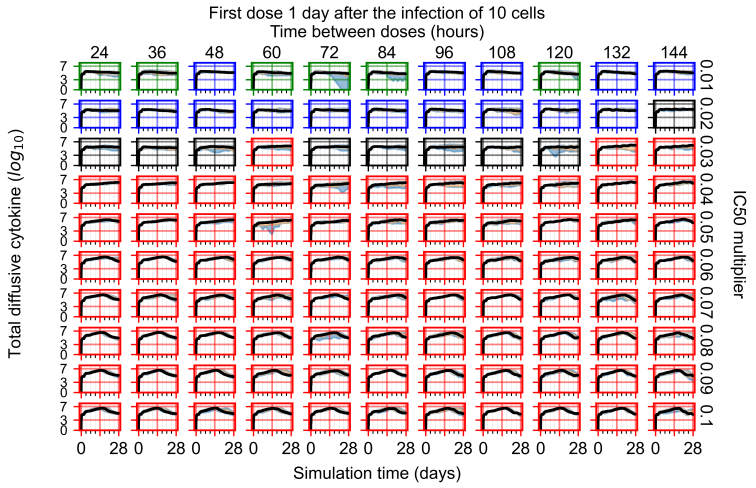


Figure A.121: Diffusive cytokine amount for 8 replicas of the treatment simulation, Y axis in log scale, exponent values as tick-marks.

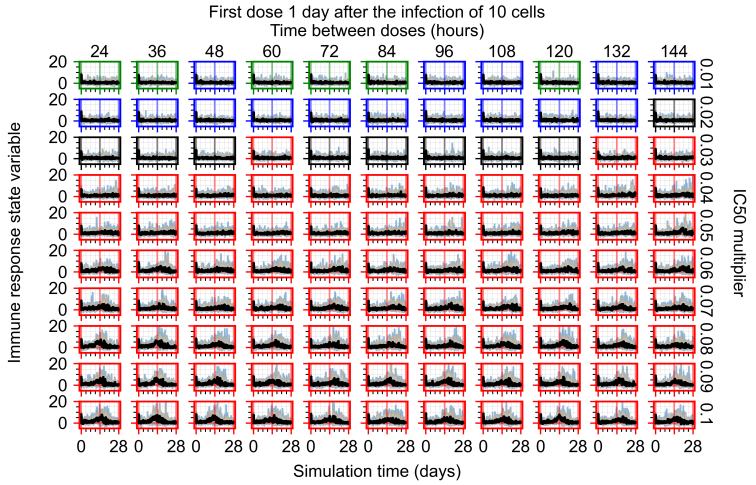
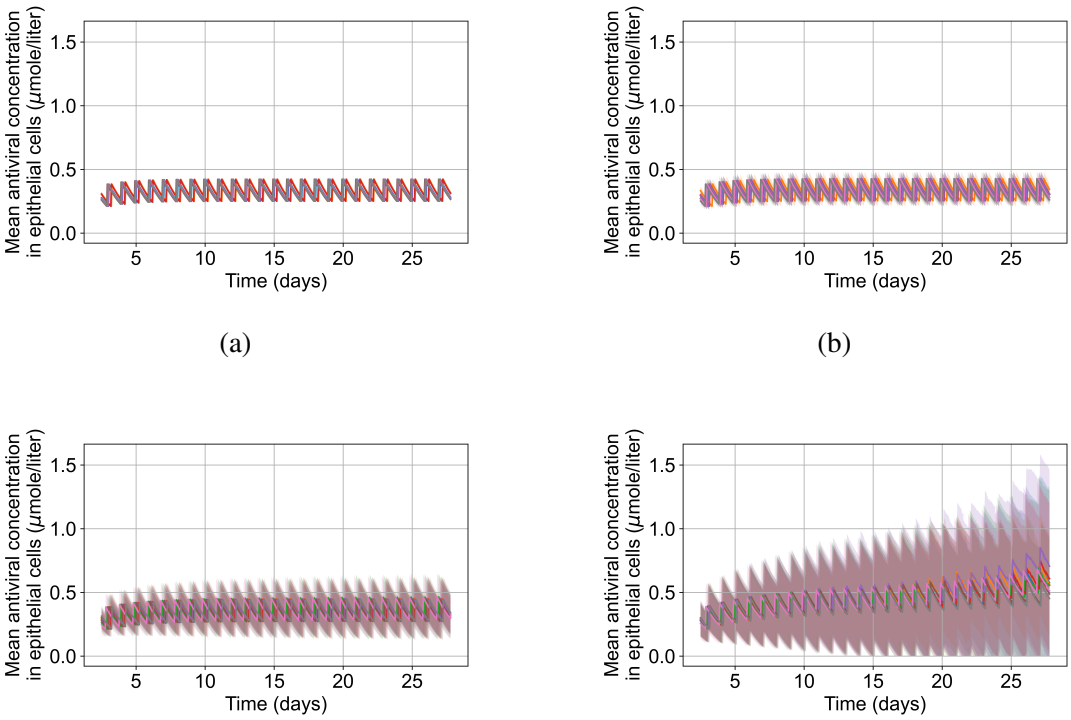


Figure A.122: Immune response state variable number. Positive values correspond to an inflammatory state, negative to an anti-inflammatory state.

A.6.5.3 How heterogeneity affects intracellular drug levels

Here we show how different levels of metabolism heterogeneity affect intra-cellular drug levels. For high heterogeneity some cells have an internal concentration of antiviral close to or at zero.



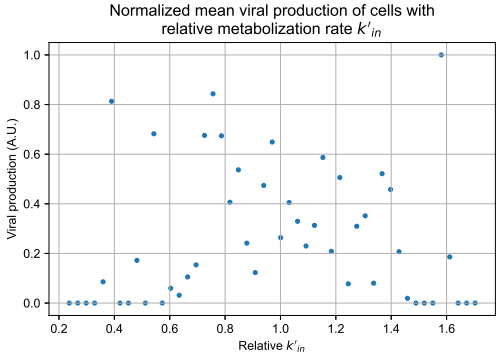
(a) (b)

(c) (d)

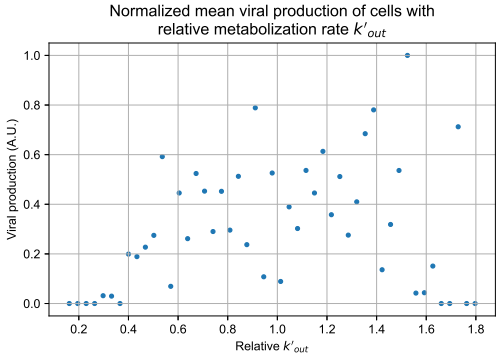
Figure A.123: Mean antiviral drug levels in virus-releasing infected cells (solid lines) with standard deviation (shaded regions) versus time for individual simulation replicates under different simulation options. Sub-figure A.123a is treated with no metabolism heterogeneity, A.123b is treated with a metabolism standard deviation of 0.1, A.123c is treated with a metabolism standard deviation of 0.25, and A.123d is treated with a metabolism standard deviation of 0.25.

A.7 Supplementary results for viral production metabolism rate correlation

For the figures of this appendix as well as Figure 3.12 we performed 4 simulation replicas for each parameter set. We group the cells of all parameter set replicas into 50 bins by their metabolism rates, we then compute the mean production of that bin and the maximum viral production of all bins. We plot the mean of the bin divided by the maximum mean production of all bins versus the metabolism rate.

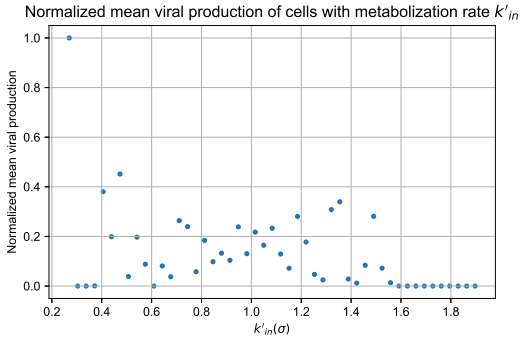


(a)

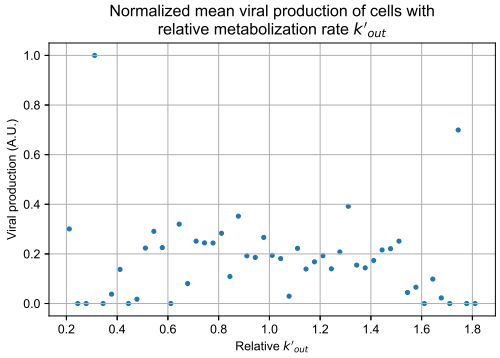


(b)

Figure A.124: Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with rapid clearance parameters. A.124a Results for simulations varying only k_{in}. A.124b Results for simulations varying only k_{out}.

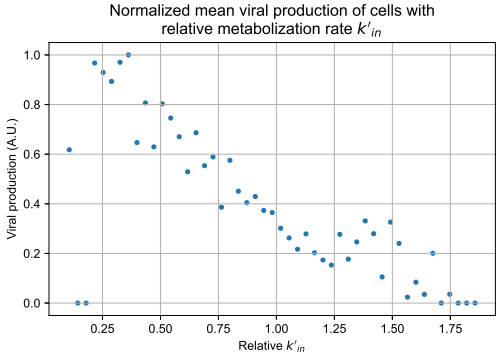


(a)

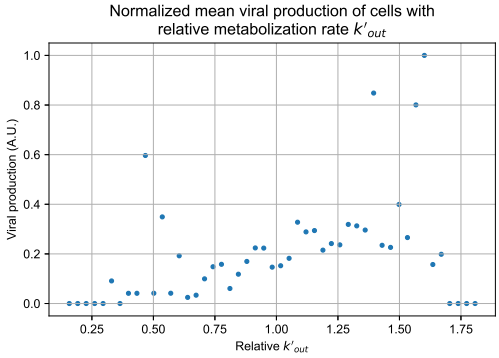


(b)

Figure A.125: Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with rapid clearance parameters. A.125a Results for simulations varying only k_{in}. A.125b Results for simulations varying only k_{out}.

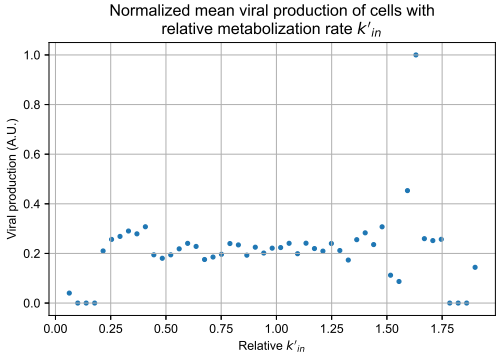


(a)

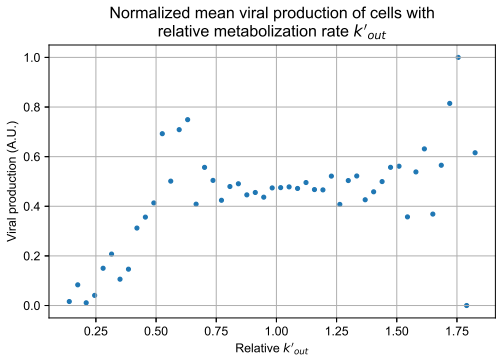


(b)

Figure A.126: Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with slow clearance parameters. A.126a Results for simulations varying only k_{in}. A.126b Results for simulations varying only k_{out}.



(a)



(b)

Figure A.127: Mean viral production of cells versus their metabolism rates normalized by the maximum mean production with widespread infection parameters. A.127a Results for simulations varying only k_{in}. A.127b Results for simulations varying only k_{out}.

APPENDIX B

TRANSLATING MODEL SPECIFICATIONS APPENDIX

B.1 Converting Space Dimensions

```
def get_dims(pcdict, space_convs=_space_convs):
    """
    Parses PhysiCell data and generates CC3D space dimensions and unit
                                        conversions

    This function looks for the value of the maximum and minimum of all
                                        coordinates in PhysiCell (
    pcdict['domain']['x_min'], pcdict['domain']['x_max'], etc) and saves
                                         them to variables. It also
                                        looks for the
    discretization variables from PhysiCell (pcdict['domain']['dx'], etc
                                        ) and saves them to variables.
                                        Using the size of
    the domain and the discretization it defines what will be the number
                                         of pixels in CompuCell3D's
                                        domain.
    It also looks for the unit used in PhysiCell to determine what will
                                        be the pixel/unit factor in CC3D
                                        .

    :param pcdict: Dictionary created from parsing PhysiCell XML
    :param space_convs: Dictionary of predefined space units
    :return pcdims, ccdims: Two tuples representing the dimension data
                                        from PhysiCell and in CC3D.
          ((xmin, xmax), (ymin, ymax), (zmin, zmax), units), and
```

```
        (cc3dx, cc3dy, cc3dz, cc3dspaceunitstr, cc3dds,
                                                autoconvert_space)
    """
    xmin = float(pcdict['domain']['x_min']) if "x_min" in pcdict['domain
                                            '].keys() else None
    xmax = float(pcdict['domain']['x_max']) if "x_max" in pcdict['domain
                                            '].keys() else None

    ymin = float(pcdict['domain']['y_min']) if "y_min" in pcdict['domain
                                            '].keys() else None
    ymax = float(pcdict['domain']['y_max']) if "y_max" in pcdict['domain
                                            '].keys() else None

    zmin = float(pcdict['domain']['z_min']) if "z_min" in pcdict['domain
                                            '].keys() else None
    zmax = float(pcdict['domain']['z_max']) if "z_max" in pcdict['domain
                                            '].keys() else None

    units = pcdict['overall']['space_units'] if 'overall' in pcdict.keys
                                            () and \
                                                'space_units' in pcdict[
```

```
autoconvert_space = True
if units not in space_convs.keys():
    message = f"WARNING: {units} is not part of known space units.
                                        Automatic space-unit
                                        conversion disabled." \
              f"Available units for auto-conversion are:\n{
                                                space_convs.keys()
                                                }"
    warnings.warn(message)
    autoconvert_space = False

# the dx/dy/dz tags mean that for every voxel there are dx space-
                                        units.
# therefore [dx] = [space-unit/voxel]. Source: John Metzcar

dx = float(pcdict['domain']['dx']) if "dx" in pcdict['domain'].keys
                                        () else 1
dy = float(pcdict['domain']['dy']) if "dy" in pcdict['domain'].keys
                                        () else 1
dz = float(pcdict['domain']['dz']) if "dz" in pcdict['domain'].keys
                                        () else 1

# print(dx, dy, dz, type(dx), type(dy), type(dz), )
if not dx == dy == dz:
    message = "WARNING! Physicell's dx/dy/dz are not all the same: "
                                            \
              f"dx={dx}, dy={dy}, dz={dz}\n" \
              f"Using {max(min(dx, dy, dz), 1)}"
    warnings.warn(message)
```

```
    ds = max(min(dx, dy, dz), 1)

    diffx = 1 if xmin is None or xmax is None else round(xmax - xmin)
    diffy = 1 if ymin is None or ymax is None else round(ymax - ymin)
    diffz = 1 if zmin is None or zmax is None else round(ymax - zmin)

    cc3dx = round(diffx / ds)
    cc3dy = round(diffy / ds)
    cc3dz = round(diffz / ds)

    cc3dds = cc3dx / diffx  # pixel/unit

    # [cc3dds] = pixel/unit
    # [cc3dds] * unit = pixel

    cc3dspaceunitstr = f"1 pixel = {cc3dds} {units}"

    pcdims, ccdims = ((xmin, xmax), (ymin, ymax), (zmin, zmax), units),
                                          \
                     (cc3dx, cc3dy, cc3dz, cc3dspaceunitstr, cc3dds,
                                                             autoconvert_space
                                                             )


    return pcdims, ccdims
```

B.2 Converting Time Dimensions

B.3 Converting Cell Types and Extracting Mechanics

B.3.1 Cell Types Extraction

```
def make_cell_type_plugin(pcdict):
    """
```

```
    Makes the cell type plugin for CC3D

    Passes pcdict to make_cell_type_tags to generate the cell types and
                                        returns the results

    :param pcdict: Dictionary created from parsing PhysiCell XML
    :return ct_str, wall, cell_types: string setting the cell type
                                        plugin for cc3d's XML, bool for
                                        the presence of a
    wall cell type, list of cell types
    """
    ct_str = '\n<Plugin Name="CellType">\n\t' \
             '<CellType TypeId="0" TypeName="Medium"/>\n'
    typesstr, wall, cell_types = make_cell_type_tags(pcdict)
    ct_str += typesstr
    ct_str += '</Plugin>'
    return ct_str, wall, cell_types
```

```
def make_cell_type_tags(pcdict):
    """
    Parses the PhysiCell dictionary to fetch the cell type names,
                                        generates the internal part of
                                        the cell type plugin
    :param pcdict: Dictionary created from parsing PhysiCell XML
    :return s, create_wall, cell_types: string for the cell type plugin,
                                         bool for the existance of a
                                        wall cell type,
    list of cell type names
    """
    s = ''
    cell_types = []
    idx = 1
    for child in pcdict['cell_definitions']['cell_definition']:
```

```
        # print(child.tag, child.attrib, child.text)
        name = child['@name'].replace(" ", "_")
        cell_types.append(name)
        ctt = f'\t<CellType TypeId="{idx}" TypeName="{name}"/>\n'
        s += ctt
        idx += 1
    create_wall = get_boundary_wall(pcdict)
    if create_wall:
        s += f'\t<CellType Freeze="" TypeId="{idx}" TypeName="WALL"/>\n'
        cell_types.append("WALL")
    return s, create_wall, cell_types
```

B.3.2 Mechanics Extraction

```
def get_cell_constraints(pcdict, space_unit, minimum_volume=8):
    """
    Extracts cell constraints from the given PhysiCell pcdict.

    Parameters:
    -----------
    pcdict : dict
        Dictionary created from parsing PhysiCell XML. Must contain a "
                                            cell_definitions" key that
                                            maps
        to a dictionary with a "cell_definition" key. This key should
                                            contain a list of
                                            dictionaries, each of which
        represents a Cell Type.
    space_unit : float
        A scaling factor for the simulation's spatial units. All volumes
                                             extracted from pcdict will
                                            be multiplied by
```

```
    this factor raised to the power of the dimensionality of the
                                        simulation space.
minimum_volume : float, optional
    The minimum volume allowed for any cell in pixels. If a cell's
                                        volume falls below this
                                        threshold after scaling,
    the translator will reconvert space so that the minimum cell
                                        volume is  equal to this
                                        threshold. Defaults to 8.

Returns:
--------
constraints : dict
    A dictionary containing the constraints for each Cell Type found
                                         in pcdict. Each key is a
                                        Cell Type name
    (converted to an underscore-delimited string), and each value is
                                         a dictionary containing
                                        information about
    that Cell Type's volume, mechanics, custom data, and phenotypes.
any_below : bool
    A boolean indicating whether any cells had volumes that fell
                                        below minimum_volume after
                                        scaling.
volumes : list
    A list containing the scaled volumes of each Cell Type found in
                                        pcdict.
minimum_volume : float
    The minimum volume allowed for any cell, after scaling.

Raises:
-------
UserWarning
```

```
    If a Cell Type's volume is missing a unit or value in pcdict, or
                                            if the scaled volume falls
                                           below
    minimum_volume.
"""
constraints = {}
any_below = False
volumes = []
for child in pcdict['cell_definitions']['cell_definition']:
    ctype = child['@name'].replace(" ", "_")
    constraints[ctype] = {}
    volume, unit = get_cell_volume(child)
    if volume is None or unit is None:
        message = f"WARNING: cell volume for cell type {ctype}
                                               either doesn't have a
                                               unit \n(unit found: {
                                               unit}) " \
                  f"or" \
                  f" doesn't have a value (value found: {volume}). \
                                                    nSetting the
                                                    volume to be
                                                    the minimum
                                                    volume, " \
                  f"{minimum_volume}"
        warnings.warn(message)
        volume = None
        unit = "not specified"
        dim = 3
    else:
        dim = int(unit.split("^")[-1])
    if volume is None:
        volumepx = None
    else:
```

```
            volumepx = volume * (space_unit ** dim)
        below, minimum_volume = check_below_minimum_volume(volumepx,
                                                           minimum=minimum_volume)
        constraints[ctype]["volume"] = {f"volume ({unit})": volume,
                                        "volume (pixels)": volumepx}
        volumes.append(volumepx)
        if below:
            any_below = True
            message = f"WARNING: converted cell volume for cell type {
                                                 ctype} is below {
                                                 minimum_volume}.
                                                 Converted volume " \
                      f"{volumepx}. \nIf cells are too small in CC3D
                                                       they do not
                                                       behave in a
                                                       biological
                                                       manner and may
                                                        " \
                      f"disapear. \nThis program will enforce that: 1)
                                                       the volume
                                                       proportions
                                                       stay as before
                                                       ; 2) the " \
                      f"lowest cell volume is {minimum_volume}"
            warnings.warn(message)
        constraints[ctype]["mechanics"] = get_cell_mechanics(child)
        constraints[ctype]["custom_data"] = get_custom_data(child)
        constraints[ctype]["phenotypes"], constraints[ctype]["
                                           phenotypes_names"] =
                                           get_cell_phenotypes(child)
    return constraints, any_below, volumes, minimum_volume
```

B.4 Converting Secretion and Uptake rates

```
def convert_secretion_uptake_data(sec_dict, time_conv, pctimeunit):
"""
Convert secretion data from PhysiCell to CompuCell3D format.

This function converts the secretion data parsed from PhysiCell to
                                    CompuCell3D python commands to
                                    perform
secretion. It also adds comment to indicate any potential
                                    discrepancies between the two
                                    formats, such as
differences in the handling of target secretion or uptake bounds.
                                    The resulting dictionary is
                                    returned.

Parameters
----------
sec_dict : dict
    Dictionary containing parsed secretion data from PhysiCell.
time_conv : float
    Conversion factor for time units.
pctimeunit : str
    PhysiCell time unit.

Returns
-------
dict
    Dictionary containing CompuCell3D secretion python commands.
"""
if not sec_dict:
    return {}
new_sec_dict = sec_dict
```

```
    secretion_comment = '#WARNING: PhysiCell has a concept of "target
                                        secretion" that CompuCell3D does
                                         not. \n#The ' \
                        'translating program attempts to implement it,
                                                        but it may
                                                        not be a 1
                                                        to 1
                                                        conversion.'
    uptake_comment = '#WARNING: To avoid negative concentrations, in
                                        CompuCell3D uptake is "bounded."
                                         \n# If the amount' \
                     ' that would be uptaken is larger than the value at
                                                         that pixel,\n#
                                                         the uptake
                                                        will be a set
                                                        ratio ' \
                     'of the amount available.\n# The conversion program
                                                         uses 1 as the
                                                        ratio,\n# you
                                                        may want to ' \
                     'revisit this.'
    for ctype in sec_dict.keys():
        type_sec = sec_dict[ctype]
        new_type_sec = type_sec
        for field, data in type_sec.items():
            unit = data['secretion_unit'] if 'secretion_unit' in data.
                                                    keys() else None
            mcs_secretion_rate, extra_sec_comment =
                                                    convert_secretion_rate(
                                                    data['secretion_rate'],
                                                    unit,
                time_conv,pctimeunit)
            data['secretion_rate_MCS'] = mcs_secretion_rate
```

```
            data['secretion_comment'] = secretion_comment +
                                                extra_sec_comment
            unit = data['net_export_unit'] if 'net_export_unit' in data.
                                                keys() else None
            mcs_net_secretion_rate, extra_net_sec_comment =
                                                convert_net_secretion(
                                                data['net_export'], unit
                                                ,
                time_conv, pctimeunit)
            data['net_export_MCS'] = mcs_net_secretion_rate
            data['net_secretion_comment'] = extra_net_sec_comment
            mcs_uptake_rate, extra_up_comment = convert_uptake_rate(data
                                                ['uptake_rate'], data['
                                                uptake_unit'],
                time_conv, pctimeunit)
            data['uptake_rate_MCS'] = mcs_uptake_rate
            data['uptake_comment'] = uptake_comment + extra_up_comment
            if "secretion_target" not in data.keys():
                data["secretion_target"] = 0
            new_type_sec[field] = data
        new_sec_dict[ctype] = new_type_sec
    return new_sec_dict
```

B.5 Re-scaling time

```
def reconvert_time_parameter(d_elements, cctime, max_D=50):
    """
    Parameters:
    -----------
    d_elements : dict
        a dictionary of diffusion elements.
    cctime : tuple
```

```
    The previously converted time unit parameters
max_D : float, optional
    Maximum diffusion constant allowed. Default 50
Returns:
-------
    d_elements : dict
        the reconverted dictionary of diffusion elements.
    new_cctime : tuple
        the reconverted tuple of cc3d time parameters
"""
# get_diffusion_constants returns a list of diffusion constants that
                                     do not use the steady-state
                                     solver
diffusion_constants = get_diffusion_constants(d_elements)
if len(diffusion_constants) == 0 or not max(diffusion_constants) >
                                     max_D:
    return d_elements, cctime
message = "WARNING: the converted diffusion parameters were very
                                     high, using them as is would
                                     result in a very " \
          "slow simulation. The translating software will reconvert
                                              the time unit in order
                                               to keep the diffusion
                                              " \
          " parameters low."
warnings.warn(message)
max_old_D = max(diffusion_constants)
reduction_proportion = round(0.9 * max_D / max_old_D, 2)
new_gammas = []
for key in d_elements.keys():
    d_elements[key]["D"] *= reduction_proportion
    d_elements[key]["D_conv_factor"] *= reduction_proportion
    d_elements[key]["D_conv_factor_text"] = \
```

```
            d_elements[key]["D_conv_factor_text"].split("=")[0] + \
            " = " + \
            f'{d_elements[key]["D_conv_factor"]} ' + \
            d_elements[key]["D_conv_factor_text"].split(" ")[-1]

        d_elements[key]["gamma"] *= reduction_proportion
        d_elements[key]["gamma_conv_factor"] *= reduction_proportion
        d_elements[key]["gamma_conv_factor_text"] = \
            d_elements[key]["gamma_conv_factor_text"].split("=")[0] + \
            " = " + \
            f'{d_elements[key]["gamma_conv_factor"]} ' + \
            d_elements[key]["gamma_conv_factor_text"].split(" ")[-1]

        new_gammas.append(d_elements[key]["gamma"])
    new_cctime = [min(int(cctime[0] / reduction_proportion), 10 ** 9),
                  f'1 MCS = {cctime[2] * reduction_proportion} {cctime[1
                                                          ].split(" ")[-1]}'
                                                          ,
                  cctime[2] * reduction_proportion,
                  cctime[3]]
    return d_elements, new_cctime
```

B.6 Initial conditions

```
def default_initial_cell_config(celltypes, xmax, ymax, zmax):
    """
    Returns the default UniformInitializer steppable.

    The default_initial_cell_config function returns an XML string with
                                        a configuration for
    the UniformInitializer steppable in CompuCell3D. The configuration
                                        is based on the input parameters
```

```
                                        of celltypes,
    xmax, ymax, and zmax.

    The UniformInitializer steppable is responsible for initializing the
                                         initial configuration of cells
                                        in the
    simulation. This function sets up a rectangular slab of cells with a
                                         specified width and gap size,
                                        and restricts
    the cell types to those specified in the celltypes list. The WALL
                                        cell type is excluded from the
                                        initialization
    process.

    Parameters
    ----------
    celltypes : list of str
        List of cell types.
    xmax : int
        Maximum x dimension of the simulation.
    ymax : int
        Maximum y dimension of the simulation.
    zmax : int
        Maximum z dimension of the simulation.

    Returns
    -------
    str
        Configured XML string.
    """
    beg = '''<Steppable Type="UniformInitializer">
\t<!-- Initial layout of cells in the form of rectangular slab -->
```

```
\t<!-- PhysiCell has many complex ways of defining the initial
                                    arangement -->
\t<!-- of cells. By default the translator uses a simple configuration,
                                    -->
\t<!-- you are responsible for analysing the initialization of the
                                    original -->
\t<!-- model and reimplement it accordingly -->
\t<Region>\n'''
    if (zmax != 1 and zmax != 0) and (xmax > 10 and ymax > 10 and zmax >
                                        10):
        box_min = f'\t\t<BoxMin x="{10}" y="{10}" z="{10}"/>\n'
        box_max = f'\t\t<BoxMax x="{xmax - 10}" y="{ymax - 10}" z="{zmax
                                        - 10}"/>\n'
    elif zmax != 1 and zmax != 0:
        box_min = f'\t\t<BoxMin x="{1}" y="{1}" z="{1}"/>\n'
        box_max = f'\t\t<BoxMax x="{xmax - 1}" y="{ymax - 1}" z="{zmax -
                                        1}"/>\n'
    else:
        box_min = f'\t\t<BoxMin x="{10}" y="{10}" z="{0}"/>\n'
        box_max = f'\t\t<BoxMax x="{xmax - 10}" y="{ymax - 10}" z="{1}
                                        "/>\n'

    gap = "\t\t<Gap>0</Gap>\n\t\t<Width>7</Width>\n"

    types = ''
    for t in celltypes:
        if t.upper() == "WALL":
            continue
        types += f"{t},"

    types = types[:-1]

    types = "\t\t<Types>" + types + "</Types>\n"
```

```
    end = '''\t</Region>
</Steppable>'''
    steppable_string = beg + box_min + box_max + gap + types + end
    return steppable_string
```

B.7 Generating diffusion's XML

```
def make_diffusion_plug(diffusing_elements, celltypes, flag_2d):
    """
    Generates the XML for the diffusion steppables in CC3D a diffusion
                                        plug, combining the finite
                                        element (FE) solver
    and the steady-state solver.

    Parameters
    ----------
    diffusing_elements : dict
        Dictionary of diffusing elements and their parameters
        Each key in the dictionary represents a diffusing element and
                                            its value is a dictionary
                                            with the following keys:
        - use_steady_state : bool
            Whether or not to use steady-state diffusion solver for this
                                                element.
        - concentration_units : str
            The concentration units of this element.
        - D_w_units : float
            The diffusion constant with units of concentration^2/time.
        - D_og_unit : str
            The original unit of the diffusion constant.
        - D : float
```

```
        The diffusion constant without units.
    - gamma_w_units : float
        The decay constant with units of 1/time.
    - gamma_og_unit : str
        The original unit of the decay constant.
    - gamma : float
        The decay constant without units.
    - initial_condition : str
        The initial concentration expression for this element.
    - dirichlet : bool
        Whether or not to use Dirichlet boundary conditions for this
                                                element.
    - dirichlet_value : float
        The value of Dirichlet boundary condition for this element.
celltypes : list
    List of cell types
flag_2d : bool
    Whether the simulation is in 2D

Returns
-------
str
    The combined string of the FE and steady-state solvers
"""

use_regular, use_steady = determine_diffusion_existence(
                                      diffusing_elements)

if use_regular:
    FE_solver = make_diffusion_FE(diffusing_elements, celltypes,
                                          flag_2d)
else:
    FE_solver = ""
```

```
    if use_steady:
        steady_state_solver = make_diffusion_steady(diffusing_elements,
                                                    flag_2d)
    else:
        steady_state_solver = ""

    return FE_solver + steady_state_solver
```

```
def determine_diffusion_existence(diffusing_elements):
    steadys = []
    for _, item in diffusing_elements.items():
        steadys.append(item["use_steady_state"])
    if not bool(len(steadys)):
        return False, False
    steady = any(steadys)
    regular = any([not el for el in steadys])
    return regular, steady
```

```
def make_diffusion_FE(diffusing_elements, celltypes, flag_2d):
    """
    Converts a dictionary of diffusion properties into a CC3D
                                            DiffusionSolverFE XML
                                            configuration string. T

    This function generates an XML string that can be used to configure
                                            CC3D's DiffusionSolverFE. It
                                            takes three
    arguments: diffusing_elements, celltypes, and flag_2d.
                                            diffusing_elements is a
                                            dictionary of the diffusing
    elements, where each key is the name of the diffusing element and
                                            the corresponding value is
                                            another dictionary
```

```
    containing the properties of that element, such as the diffusion
                                        constant and initial
                                        concentration. celltypes is a
    list of the cell types, and flag_2d is a boolean indicating whether
                                        the simulation is in two
                                        dimensions or not.

    The function loops through each diffusing element in the dictionary
                                        and generates a string with
                                        information about
    the diffusion field, including its name, diffusion data (such as
                                        diffusion and decay constants),
                                        initial
    concentration, and boundary conditions. It then concatenates these
                                        strings together to create the
                                        full XML string.

    Parameters
    ----------
    diffusing_elements : dict
        A dictionary of diffusion properties. Each key represents a
                                            diffusing element and
                                            contains a nested dictionary
        with keys "D", "gamma", "concentration_units", "D_w_units", "
                                            D_og_unit", "gamma_w_units",
                                             "gamma_og_unit",
        "use_steady_state", "initial_condition", "dirichlet", "
                                            dirichlet_value"
    celltypes : list
        A list of cell types.
    flag_2d : bool
        A boolean value indicating whether the simulation is in 2D or 3D
                                            .
```

```
    Returns
    -------
    str
        A string representation of the DiffusionSolverFE XML
                                                configuration file.

    """
    header = f'\n\n\t<Steppable Type="DiffusionSolverFE">\n\t\t<!-- The
                                        conversion uses
                                        DiffusionSolverFE and' \
             f' SteadyStateDiffusionSolver ' \
             f'by default. You may ' \
             f'wish to use another diffusion solver-->\n'

    full_str = header

    for key, item in diffusing_elements.items():

        if item["use_steady_state"]:
            continue

        name = key.replace(" ", "_")

        # diffusion data
        df_str = f'\t\t<DiffusionField Name="{name}">\n\t\t\t<
                                            DiffusionData>\n\t\t\t\t<
                                            FieldName>{name}</FieldName
                                            >\n'
        conc_units = f'\t\t\t\t<Concentration_units>{item["
                                            concentration_units"]}</
                                            Concentration_units>\n'
        og_D = f'\t\t\t\t<Original_diffusion_constant D="{item["
```

```
                                        D_w_units"]}" units= "{item
                                        ["D_og_unit"]}"/>\n'
# conv = f'\t\t\t\t<CC3D_to_original units="(pixel^2/MCS)/(item
                                        ["D_og_unit"])">{item["
                                        D_conv_factor"]}' \
#         '</CC3D_to_original>'
D_str = f'\t\t\t\t<GlobalDiffusionConstant>{item["D"]}</
                                        GlobalDiffusionConstant>\n'
og_g = f'\t\t\t\t<Original_decay_constant gamma="{item["
                                        gamma_w_units"]}" units= "{
                                        item["gamma_og_unit"]}"/>\n'
g_str = f'\t\t\t\t<GlobalDecayConstant>{item["gamma"]}</
                                        GlobalDecayConstant>\n'

init_cond_warn = '\t\t\t\t<!-- CC3D allows for diffusing fields
                                        initial conditions, if one
                                        was detected it ' \
                 'will -->\n' \
                 '\t\t\t\t<!-- be used here. For several reasons
                                                        it may not
                                                        work, if
                                                        something
                                                        looks wrong
                                                        with' \
                 ' -->\n' \
                 '\t\t\t\t<!-- your diffusing field at the start
                                                        of the
                                                        simulation
                                                        this may be
                                                        the reason
                                                        .' \
                 ' -->\n' \
                 '\t\t\t\t<!-- CC3D also allows the diffusing
```

```
                                                    field
                                                    initial
                                                    condition
                                                    to be set
                                                    by a file.
                                                    ' \
                    'Conversion of a -->\n' \
                    '\t\t\t\t<!-- PhysiCell diffusing field initial
                                                     condition
                                                    file into a
                                                     CC3D
                                                    compliant
                                                    one is ' \
                    'left as -->\n' \
                    '\t\t\t\t<!-- an exercise to the reader. -->\n'

init_cond = f'\t\t\t\t <InitialConcentrationExpression>{item["
                                        initial_condition"]}<' \
    f'/InitialConcentrationExpression>' \
    '\n\t\t\t\t<!-- <ConcentrationFileName>INITIAL CONCENTRATION
                                         FIELD - typically a
                                        file with ' \
    'path Simulation/NAME_OF_THE_FILE.txt</ConcentrationFileName
                                        > -->'


het_warning = "\n\t\t\t\t<!-- CC3D allows the definition of D
                                        and gamma on a cell type
                                        basis: -->\n"
cells_str = ""
for t in celltypes:
    cells_str += f'\t\t\t\t<!--<DiffusionCoefficient CellType="{
                                        t}">{item["D"]}</
                                        DiffusionCoefficient
```

```
                                        >-->\n'
    cells_str += f'\t\t\t\t<!--<DecayCoefficient CellType="{t}">
                                        {item["gamma"]}</
                                        DecayCoefficient>-->\n'
close_diff_data = "\t\t\t</DiffusionData>\n"

# boundary conditions

bc_head = '\t\t\t<BoundaryConditions>\n\t\t\t\t<!-- PhysiCell
                                    has either Dirichlet
                                    boundary conditions (i.e. '
                                    \
          'constant ' \
          'value) -->\n\t\t\t\t<!-- or "free floating" boundary
                                            conditions (i.e.,
                                            constant flux = 0)
                                            . -->' \
          '\n\t\t\t\t<!-- CC3D ' \
          'allows ' \
          'for more control of boundary conditions, you may want
                                             to revisit the
                                            issue. -->\n'
if item['dirichlet']:
    bc_body = f'\t\t\t\t<Plane Axis="X">\n\t\t\t\t\t<
                                        ConstantValue
                                        PlanePosition="Min"
                                        Value=' \
              f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                                ConstantValue
                                                PlanePosition
                                                ="Max" Value='
                                                 \
              f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!--
```

```
                                        Other options
                                        are (examples)
                                        : -->\n\t\t\t\
                                        t\t' \
                    f'<!--<ConstantDerivative PlanePosition="Min"
                                        Value="10.0"/>
                                         -->\n\t\t\t\t
                                        \t<!--' \
                    f'<ConstantDerivative PlanePosition="Max" Value="
                                        10.0"/> -->\n\
                                        t\t\t\t\t<!--<
                                        Periodic/>-->'
                                         \
                    '\t\t\t\t</Plane>\n' \
                    f'\t\t\t\t<Plane Axis="Y">\n\t\t\t\t\t<
                                        ConstantValue
                                        PlanePosition
                                        ="Min" Value='
                                         \
                    f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                        ConstantValue
                                        PlanePosition
                                        ="Max" Value='
                                         \
                    f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!--
                                        Other options
                                        are (examples)
                                        : -->\n\t\t\t\
                                        t\t' \
                    f'<!--<ConstantDerivative PlanePosition="Min"
                                        Value="10.0"/>
                                         -->\n\t\t\t\t
                                        \t<!--' \
```

```
                f'<ConstantDerivative PlanePosition="Max" Value="
                                                    10.0"/> -->\n\
                                                    t\t\t\t\t<!--<
                                                    Periodic/>-->'
                                                     \
                '\n\t\t\t\t</Plane>\n'
        if not flag_2d:
            bc_body += f'\t\t\t\t<Plane Axis="Z">\n\t\t\t\t\t<
                                                ConstantValue
                                                PlanePosition="Min"
                                                Value=' \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                                    ConstantValue
                                                    PlanePosition="
                                                    Max" Value=' \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!-- Other
                                                     options are (
                                                    examples): -->\n\
                                                    t\t\t\t\t' \
                f'<!--<ConstantDerivative PlanePosition="Min" Value="
                                                    10.0"/> -->\n\t\t
                                                    \t\t\t<!--' \
                f'<ConstantDerivative PlanePosition="Max" Value="10.0
                                                    "/> -->\n\t\t\t\t
                                                    \t<!--<Periodic
                                                    />-->' \
                '\n\t\t\t\t</Plane>\n'
    else:
        bc_body = f'\t\t\t\t<Plane Axis="X">\n\t\t\t\t\t<
                                            ConstantDerivative
                                            PlanePosition="Min"
                                            Value=' \
          f'"0"/>\n\t\t\t\t\t<ConstantDerivative PlanePosition="Max"
```

```
                                        Value=' \
    f'"0"/>\n\t\t\t\t\t<!-- Other options are (examples): -->\
                                        n\t\t\t\t\t' \
    f'<!--<ConstantValue PlanePosition="Min" Value="10.0"/>
                                        -->\n\t\t\t\t\t<!--' \
    f'<ConstantValue PlanePosition="Max" Value="10.0"/> -->\n\
                                        t\t\t\t\t<!--<Periodic
                                        />-->' \
    '\t\t\t\t</Plane>\n' \
    f'\t\t\t\t<Plane Axis="Y">\n\t\t\t\t\t<ConstantDerivative
                                        PlanePosition="Min"
                                        Value=' \
    f'"0"/>\n\t\t\t\t\t<ConstantDerivative PlanePosition="Max"
                                         Value=' \
    f'"0"/>\n\t\t\t\t\t<!-- Other options are (examples): -->\
                                        n\t\t\t\t\t' \
    f'<!--<ConstantValue PlanePosition="Min" Value="10.0"/>
                                        -->\n\t\t\t\t\t<!--' \
    f'<ConstantValue PlanePosition="Max" Value="10.0"/> -->\n\
                                        t\t\t\t\t<!--<Periodic
                                        />-->' \
    '\t\t\t\t</Plane>\n'
if not flag_2d:
    bc_body += f'\t\t\t\t<Plane Axis="Z">\n\t\t\t\t\t<
                                          ConstantDerivative
                                          PlanePosition="Min"
                                          Value=' \
        f'"0"/>\n\t\t\t\t\t<ConstantDerivative PlanePosition
                                             ="Max" Value=' \
        f'"0"/>\n\t\t\t\t\t<!-- Other options are (examples):
                                              -->\n\t\t\t\t\t'
                                              \
        f'<!--<ConstantDerivative PlanePosition="Min" Value="
```

```
                                                    10.0"/> -->\n\t\t
                                                    \t\t\t<!--' \
                    f'<ConstantDerivative PlanePosition="Max" Value="0.0
                                                    "/> -->\n\t\t\t\t
                                                    \t<!--<Periodic
                                                    />-->' \
                    '\t\t\t\t</Plane>\n'
        close_bc = "</BoundaryConditions>\n"
        close_field = "</DiffusionField>\n"

        full_field_def = df_str + conc_units + og_D + D_str + og_g +
                                              g_str + init_cond_warn +
                                              init_cond + het_warning +
                                              cells_str + \
                         close_diff_data + bc_head + bc_body + close_bc
                                                                    +
                                                                    close_field
        full_str += full_field_def
    full_str += "</Steppable>\n"
    return full_str
```

```
def make_diffusion_steady(diffusing_elements, flag_2d):
    """
    Creates a steady-state diffusion solver configuration for CC3D
                                          simulations based on the given
                                          diffusing elements.

    This function generates an XML string that can be used to configure
                                          CC3D's steady state diffusion
                                          solver. It takes
    two arguments: diffusing_elements and flag_2d. diffusing_elements is
                                           a dictionary of the diffusing
    elements, where each key is the name of the diffusing element and
```

```
                                    the corresponding value is
                                    another dictionary
containing the properties of that element, such as the diffusion
                                    constant and initial
                                    concentration. flag_2d is a
boolean indicating whether the simulation is in two dimensions or
                                    not.

The function loops through each diffusing element in the dictionary
                                    and generates a string with
                                    information about
the diffusion field, including its name, diffusion data (such as
                                    diffusion and decay constants),
                                    initial
concentration, and boundary conditions. It then concatenates these
                                    strings together to create the
                                    full XML string.

Parameters
----------
diffusing_elements : dict
    diffusing_elements : dict
    A dictionary of diffusion properties. Each key represents a
                                        diffusing element and
                                        contains a nested dictionary
    with keys "D", "gamma", "concentration_units", "D_w_units", "
                                        D_og_unit", "gamma_w_units",
                                         "gamma_og_unit",
    "use_steady_state", "initial_condition", "dirichlet", "
                                        dirichlet_value"

flag_2d : bool
    A boolean indicating whether to use 2D or 3D solver.
```

```
    Returns
    -------
    str
        A string containing the configuration for the steady-state
                                            diffusion solver in CC3D
                                            simulations.
    """
    if flag_2d:
        header = f'\n\n\t<Steppable Type="SteadyStateDiffusionSolver2D
                                            ">\n\t\t<!-- The conversion
                                            uses ' \
                 f'DiffusionSolverFE and' \
                 f' SteadyStateDiffusionSolver ' \
                 f'by default. You may ' \
                 f'wish to use another diffusion solver-->\n'
    else:
        header = f'\n\n\t<Steppable Type="SteadyStateDiffusionSolver">\n
                                            \t\t<!-- The conversion uses
                                             ' \
                 f'DiffusionSolverFE and' \
                 f' SteadyStateDiffusionSolver ' \
                 f'by default. You may ' \
                 f'wish to use another diffusion solver-->\n'

    full_str = header

    for key, item in diffusing_elements.items():
        if not item["use_steady_state"]:
            continue

        name = key.replace(" ", "_")
```

```
df_str = f'\t\t<DiffusionField Name="{name}">\n\t\t\t<
                                    DiffusionData>\n\t\t\t\t<
                                    FieldName>{name}</FieldName
                                    >\n'
conc_units = f'\t\t\t\t<Concentration_units>{item["
                                    concentration_units"]}</
                                    Concentration_units>\n'
og_D = f'\t\t\t\t<Original_diffusion_constant D="{item["
                                    D_w_units"]}" units= "{item
                                    ["D_og_unit"]}"/>\n'

D_str = f'\t\t\t\t<DiffusionConstant>{item["D"]}</
                                    DiffusionConstant>\n'
og_g = f'\t\t\t\t<Original_decay_constant gamma="{item["
                                    gamma_w_units"]}" units= "{
                                    item["gamma_og_unit"]}"/>\n'
g_str = f'\t\t\t\t<DecayConstant>{item["gamma"]}</DecayConstant
                                    >\n'

init_cond_warn = '\t\t\t\t<!-- CC3D allows for diffusing fields
                                    initial conditions, if one
                                    was detected it ' \
                 'will -->\n' \
                 '\t\t\t\t<!-- be used here. For several reasons
                                                    it may not
                                                    work, if
                                                   something
                                                   looks wrong
                                                    with' \
                 ' -->\n' \
                 '\t\t\t\t<!-- your diffusing field at the start
                                                    of the
                                                   simulation
```

```
                                                    this may be
                                                     the reason
                                                    .' \
                    ' -->\n' \
                    '\t\t\t\t<!-- CC3D also allows the diffusing
                                                    field
                                                    initial
                                                    condition
                                                    to be set
                                                    by a file.
                                                    ' \
                    'Conversion of a -->\n' \
                    '\t\t\t\t<!-- PhysiCell diffusing field initial
                                                     condition
                                                    file into a
                                                     CC3D
                                                    compliant
                                                    one is ' \
                    'left as -->\n' \
                    '\t\t\t\t<!-- an exercise to the reader. -->\n'

    init_cond = f'\t\t\t\t <InitialConcentrationExpression>{item["
                                        initial_condition"]}<' \
        f'/InitialConcentrationExpression>' \
        '\n\t\t\t\t<!-- <ConcentrationFileName>INITIAL CONCENTRATION
                                         FIELD - typically a
                                        file with ' \
        'path Simulation/NAME_OF_THE_FILE.txt</ConcentrationFileName
                                        > -->'

    close_diff_data = "\t\t\t</DiffusionData>\n"

    # boundary conditions
```

```
bc_head = '\t\t\t<BoundaryConditions>\n\t\t\t\t<!-- PhysiCell
                                    has either Dirichlet
                                    boundary conditions (i.e. '
                                    \
          'constant ' \
          'value) -->\n\t\t\t\t<!-- or "free floating" boundary
                                              conditions (i.e.,
                                              constant flux = 0)
                                              . -->' \
          '\n\t\t\t\t<!-- CC3D ' \
          'allows ' \
          'for more control of boundary conditions, you may want
                                               to revisit the
                                              issue. -->\n'
if item['dirichlet']:
    bc_body = f'\t\t\t\t<Plane Axis="X">\n\t\t\t\t\t<
                                        ConstantValue
                                        PlanePosition="Min"
                                        Value=' \
              f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                              ConstantValue
                                              PlanePosition
                                              ="Max" Value='
                                               \
              f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!--
                                              Other options
                                              are (examples)
                                              : -->\n\t\t\t\
                                              t\t' \
              f'<!--<ConstantDerivative PlanePosition="Min"
                                              Value="10.0"/>
                                               -->\n\t\t\t\t
```

```
                                                \t<!--' \
                f'<ConstantDerivative PlanePosition="Max" Value="
                                                10.0"/> -->\n\
                                                t\t\t\t\t<!--<
                                                Periodic/>-->'
                                                 \
                '\t\t\t\t</Plane>\n' \
                f'\t\t\t\t<Plane Axis="Y">\n\t\t\t\t\t<
                                                ConstantValue
                                                PlanePosition
                                                ="Min" Value='
                                                 \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                                ConstantValue
                                                PlanePosition
                                                ="Max" Value='
                                                 \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!--
                                                Other options
                                                are (examples)
                                                : -->\n\t\t\t\
                                                t\t' \
                f'<!--<ConstantDerivative PlanePosition="Min"
                                                Value="10.0"/>
                                                 -->\n\t\t\t\t\t
                                                \t<!--' \
                f'<ConstantDerivative PlanePosition="Max" Value="
                                                10.0"/> -->\n\
                                                t\t\t\t\t<!--<
                                                Periodic/>-->'
                                                 \
                '\n\t\t\t\t</Plane>\n'
        if not flag_2d:
```

```
            bc_body += f'\t\t\t\t<Plane Axis="Z">\n\t\t\t\t\t<
                                                ConstantValue
                                                PlanePosition="Min"
                                                Value=' \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<
                                                    ConstantValue
                                                    PlanePosition="
                                                    Max" Value=' \
                f'"{item["dirichlet_value"]}"/>\n\t\t\t\t\t<!-- Other
                                                     options are (
                                                    examples): -->\n\
                                                    t\t\t\t\t' \
                f'<!--<ConstantDerivative PlanePosition="Min" Value="
                                                    10.0"/> -->\n\t\t
                                                    \t\t\t<!--' \
                f'<ConstantDerivative PlanePosition="Max" Value="10.0
                                                    "/> -->\n\t\t\t\t
                                                    \t<!--<Periodic
                                                    />-->' \
                        '\n\t\t\t\t</Plane>\n'
    else:
        bc_body = f'\t\t\t\t<Plane Axis="X">\n\t\t\t\t\t<
                                                ConstantDerivative
                                                PlanePosition="Min"
                                                Value=' \
          f'"0"/>\n\t\t\t\t\t<ConstantDerivative PlanePosition="Max"
                                                 Value=' \
          f'"0"/>\n\t\t\t\t\t<!-- Other options are (examples): -->\
                                                n\t\t\t\t\t' \
          f'<!--<ConstantValue PlanePosition="Min" Value="10.0"/>
                                                -->\n\t\t\t\t\t<!--' \
          f'<ConstantValue PlanePosition="Max" Value="10.0"/> -->\n\
                                                t\t\t\t\t<!--<Periodic
```

```
                                        />-->' \
    '\t\t\t\t</Plane>\n' \
    f'\t\t\t\t<Plane Axis="Y">\n\t\t\t\t\t<ConstantDerivative
                                        PlanePosition="Min"
                                        Value=' \
    f'"0"/>\n\t\t\t\t\t<ConstantDerivative PlanePosition="Max"
                                         Value=' \
    f'"0"/>\n\t\t\t\t\t<!-- Other options are (examples): -->\
                                        n\t\t\t\t' \
    f'<!--<ConstantValue PlanePosition="Min" Value="10.0"/>
                                        -->\n\t\t\t\t<!--' \
    f'<ConstantValue PlanePosition="Max" Value="10.0"/> -->\n\
                                        t\t\t\t<!--<Periodic
                                        />-->' \
    '\t\t\t\t</Plane>\n'
if not flag_2d:
    bc_body += f'\t\t\t\t<Plane Axis="Z">\n\t\t\t\t<
                                        ConstantDerivative
                                        PlanePosition="Min"
                                        Value=' \
        f'"0"/>\n\t\t\t\t<ConstantDerivative PlanePosition
                                            ="Max" Value=' \
        f'"0"/>\n\t\t\t\t<!-- Other options are (examples):
                                            -->\n\t\t\t\t'
                                            \
        f'<!--<ConstantDerivative PlanePosition="Min" Value="
                                            10.0"/> -->\n\t\t
                                            \t\t<!--' \
        f'<ConstantDerivative PlanePosition="Max" Value="10.0
                                            "/> -->\n\t\t\t\t
                                            \t<!--<Periodic
                                            />-->' \
            '\t\t\t\t</Plane>\n'
```

```
        close_bc = "</BoundaryConditions>\n"
        close_field = "</DiffusionField>\n"

        full_field_def = df_str + conc_units + og_D + D_str + og_g +
                                              g_str + init_cond_warn +
                                              init_cond + \
                         close_diff_data + bc_head + bc_body + close_bc
                                                                  +
                                                                  close_field
        full_str += full_field_def

    full_str += "</Steppable>\n"
    return full_str
```

B.8 Stepabble generation helper functions

```
def steppable_declaration(step_name, mitosis=False):
    if not mitosis:
        stype = "SteppableBasePy"
    else:
        stype = "MitosisSteppableBase"
    return f"class {step_name}Steppable({stype}):\n"
```

```
def steppable_imports(user_data="", phenocell_dir=False):
    if not phenocell_dir:
        phenocell_dir = "C:\\PhenoCellPy"
    imports = '''from cc3d.cpp.PlayerPython import *\nfrom cc3d import
                                          CompuCellSetup
from cc3d.core.PySteppables import *\nimport numpy as np\n
'''
    phenocell = f'''import sys\n
# IMPORTANT: PhysiCell has a concept of cell phenotype, PhenoCellPy (
                                      https://github.com/JulianoGianlupi/
```

```
                                        PhenoCellPy)
# has a similar implementation of phenotypes. You should install
                                        PhenoCellPy to translate the
                                        Phenotypes from PhysiCell.
# Then change the default path used below with your PhenoCellPy's
                                        installation directory
sys.path.extend(['{phenocell_dir}'])
global pcp_imp
pcp_imp = False
try:
\timport PhenoCellPy as pcp
\tpcp_imp = True
except:
\tpass\n\n
user_data={user_data}\n\n
'''
    return imports+phenocell
```

```
def steppable_init(frequency, mitosis=False):
    if mitosis:
        return mitosis_init(frequency)
    return f'''\n\tdef __init__(self, frequency={frequency}):
\t\tSteppableBasePy.__init__(self,frequency)\n'''
```

```
def mitosis_init(frequency):
    return f'''\n\tdef __init__(self,frequency={frequency}):
\t\tMitosisSteppableBase.__init__(self,frequency)\n'''
```

```
def steppable_start():
    return '''
\tdef start(self):
\t\t"""
\t\tCalled before MCS=0 while building the initial simulation
\t\t"""
```

```
\t\tself.pixel_to_space = float(self.get_xml_element('pixel_to_space').
                                    cdata)  # pixel/[unit], see xml for
                                    units
\t\tself.mcs_to_time = float(self.get_xml_element('mcs_to_time').cdata)
                                     # MCS/[unit], see xml for units'''
```

```
def steppable_step():
    step = '''
\tdef step(self, mcs):
\t\t"""
\t\tCalled every frequency MCS while executing the simulation

\t\t:param mcs: current Monte Carlo step
\t\t"""\n
'''
    return step



def steppable_finish():
    finish = '''
\tdef finish(self):
\t\t"""
\t\tCalled after the last MCS to wrap up the simulation. Good place to
                                    close files and do post-processing
\t\t"""\n
'''
    return finish
```

```
def steppable_on_stop():
    stop = '''
\tdef on_stop(self):
\t\t"""
\t\tCalled if the simulation is stopped before the last MCS
```

```
\t\t"""
\t\tself.finish()\n
'''
    return stop
```

```
def _add_to_function(function, extra):
    return function+'\n'+extra+'\n'
```

```
def add_to_init(init, additional_init):
    return _add_to_function(init, additional_init)
```

```
def add_to_start(start, additional_start):
    return _add_to_function(start, additional_start)
```

```
def add_to_step(step, additional_step):
    return _add_to_function(step, additional_step)
```

```
def add_to_finish(finish, additional_finish):
    return _add_to_function(finish, additional_finish)
```

```
def add_to_on_stop(on_stop, additional_on_stop):
    return _add_to_function(on_stop, additional_on_stop)
```

```
def mitosis_update_attribute():
    update = '''
\tdef update_attributes(self):
\t\tself.parent_cell.targetVolume /= 2.0
\t\tself.clone_parent_2_child()
'''
    return update
```

B.9 Generate constraint loops helper function

```
def get_dicts_for_type(ctype, cell_dicts):
    ds = []
    for d in cell_dicts:
        if ctype in d.keys():
            ds.append(d[ctype])
    return ds
```

```
def cell_type_constraint(ctype, this_type_dicts):
    if not this_type_dicts:
        return ''
    loop = f"\t\tfor cell in self.cell_list_by_type(self.{ctype.upper()}
                                        ):\n"
    full = loop
    for cell_dict in this_type_dicts:
        for key, value in cell_dict.items():
            if key == "phenotypes" and bool(cell_dict["phenotypes"]):
                line = "\t\t\tif pcp_imp:\n"
                line += f"\t\t\t\tcell.dict['{key}']=self.phenotypes['{
                                                ctype}']\n"
                line += f"\t\t\t\tcell.dict['current_phenotype'] = cell.
                                                dict['{key}']" \
                        f"['{cell_dict['phenotypes_names'][0]}'].copy()\
                                                    n"
                line += f"\t\t\t\tcell.dict['volume_conversion'] = cell.
                                                targetVolume / \\\n"
                                                 \
                    f"\t\t\t\t\tcell.dict['current_phenotype'].
                                                    current_phase.
                                                    volume.total\n"
            elif key == "custom_data":
                line = f"\t\t\t# NOTE: you are responsible for finding
                                                how this data" \
```

```
                    f"is used in the original model\n\t\t\t# and re-
                                                        implementing
                                                        in CC3D" \
                    f"\n\t\t\tcell.dict['{key}']={value}\n"
            elif type(value) == str:
                line = f"\t\t\tcell.dict['{key}']='{value}'\n"
            elif type(value) == dict:
                clean_value = value.copy()
                to_pop = []
                for subkey in value.keys():

                    if "comment" in subkey:
                        to_pop.append(subkey)
                [clean_value.pop(p) for p in to_pop]
                line = f"\t\t\tcell.dict['{key}']={clean_value}\n"
            else:
                line = f"\t\t\tcell.dict['{key}']={value}\n"
            if key in ["volume", "surface"]:
                line += apply_CC3D_constraint(key, value)
            full += line
    return full + '\n\n'
```

B.10 Generating phenotype models initialization

```
def initialize_phenotypes(constraint_dict):
    pheno_str = "\n\t\tif pcp_imp:\n"
    pheno_str += "\t\t\tself.phenotypes = {}\n"
    for ctype, cdict in constraint_dict.items():
        if "phenotypes" in cdict.keys():
            pheno_str += f"\t\t\tdt = 1/self.mcs_to_time\n"
            pheno_str += f"\t\t\tself.phenotypes['{ctype}']" + "= {}\n"
            for phenotype, data in cdict["phenotypes"].items():
```

```
time_unit = "None"
if "rate units" in data.keys():
    time_unit = data["rate units"].split("/")[-1]
fixed = []
duration = []
for fix, dur in data["phase durations"]:
    duration.append(dur)
    if fix == "TRUE":
        ff = True
    else:
        ff = False
    fixed.append(ff)
nuclear_fluid = []
nuclear_solid = []
cyto_fluid = []
cyto_solid = []
cyto_to_nucl = []
if 'fluid fraction' not in data.keys():
    data['fluid fraction'] = [.75] * len(data["phase
                                        durations"])
# if 'fluid fraction' not in data.keys():
#     data['fluid fraction'] = [.75]*len(data["phase
                                    durations"])
if 'fluid fraction' in data.keys() and 'nuclear volume'
                                    in data.keys() and '
                                    total' in data.keys
                                    ():
    for fluid, nucl, total in zip(data['fluid fraction']
                                        , data['nuclear
                                        volume'], data['
                                        total']):
        nfl = fluid * nucl
        nuclear_fluid.append(nfl)
```

```
            nuclear_solid.append(nucl - nfl)
            cytt = total - nucl
            cytf = fluid * cytt
            cyts = cytt - cytf
            cyto_fluid.append(cytf)
            cyto_solid.append(cyts)
            cyto_to_nucl.append(cytt / (1e-16 + nucl))
    else:
        nuclear_fluid = [None] * len(data["phase durations"]
                                             )
        nuclear_solid = [None] * len(data["phase durations"]
                                             )
        cyto_fluid = [None] * len(data["phase durations"])
        cyto_solid = [None] * len(data["phase durations"])
        cyto_to_nucl = [None] * len(data["phase durations"])
    if 'calcified fraction' not in data.keys():
        data['calcified fraction'] = [0] * len(data["phase
                                             durations"])

    if 'cytoplasm biomass change rate' not in data.keys():
        data['cytoplasm biomass change rate'] = [None] * len
                                             (data["phase
                                             durations"])
    if 'nuclear biomass change rate' not in data.keys():
        data['nuclear biomass change rate'] = [None] * len(
                                             data["phase
                                             durations"])
    if 'calcification rate' not in data.keys():
        data['calcification rate'] = [None] * len(data["
                                             phase durations"
                                             ])
    if 'fluid change rate' not in data.keys():
```

```
            data['fluid change rate'] = [None] * len(data["phase
                                                  durations"])
        pheno_str += \
            f"\t\t\tphenotype = pcp.get_phenotype_by_name('{
                                                 phenotype}')\n"
        pheno_str += f"\t\t\tself.phenotypes['{ctype}']['{
                                               phenotype}'] =
                                               phenotype(dt=dt, \n\
                                               t\t\t\t" \
         f"time_unit='{time_unit}', \n\t\t\t\tfixed_durations={
                                                fixed},   " \
         f"\n\t\t\t\tphase_durations={duration}, \n\t\t\t\t" \
         f"cytoplasm_volume_change_rate={data['cytoplasm biomass
                                                 change rate']}, \n
                                                 \t\t\t\t" \
         f"nuclear_volume_change_rate={data['nuclear biomass
                                                 change rate']}, \n\
                                                 t\t\t\t" \
         f"calcification_rate={data['calcification rate']}, \n\t
                                                 \t\t\t" \
         f"calcified_fraction={data['calcified fraction']}, \n\t
                                                 \t\t\t" \
         f"target_fluid_fraction={data['fluid fraction']}, \n\t\
                                                 t\t\t" \
         f"nuclear_fluid={nuclear_fluid}, \n\t\t\t\t" \
         f"nuclear_solid={nuclear_solid}, \n\t\t\t\t" \
         f"nuclear_solid_target={nuclear_solid}, \n\t\t\t\t" \
         f"cytoplasm_fluid={cyto_fluid}, \n\t\t\t\t" \
         f"cytoplasm_solid={cyto_solid}, \n\t\t\t\t" \
         f"cytoplasm_solid_target={cyto_solid}, \n\t\t\t\t" \
         f"target_cytoplasm_to_nuclear_ratio={cyto_to_nucl}, \n\
                                                 t\t\t\t" \
         f"fluid_change_rate={data['fluid change rate']})\n"
```

```
    return pheno_str
```

APPENDIX C

PHENOCELLPY APPENDIX

C.1 PhenoCellPy Python implementation

Besides defining several methods and functions to drive PhenoCellPy, we allow users to define custom functions. The user-definable functions are: `entry_function(*args)`, `exit_function(*args)`, `arrest_function(*args)`, `user_phenotype_time_step(*args)`, `user_phase_time_step(*args)`. Which are executed at Phase entry, Phase exit, to determine if a cell should exit the Phenotype and enter senescence (see Sections 5.2.2 and 5.2.3), and at each time-step. We also allow the user to define the Phase transition function (see Section 5.2.2.1). All functions that can be user-defined in PhenoCellPy **must** accept any number of optional parameters (*i.e.*, be a "python args" function). For instance, for viral infection the end of the eclipse phase will occur when a threshold of internal viral load is reached [1], so the modeler would pass the intra-cellular viral load and the threshold as arguments to the function. Some pre-packaged Phases (*e.g.*, `NecrosisSwell`, `S`) use the custom entry function to change their Cell Volume model target volumes.

We would like to note that, for formatting reasons, we've removed the docstrings and most comments from the functions presented in this section. The actual source-code for PhenoCellPy contains docstrings for all functions and comments where necessary.

C.1.1 Cellular Phase

The Phase is the "base unit" of the Phenotype. Each Phase has as attributes a descriptive name (*e.g.*, S, G, M, necrotic swelling), an index (*i.e.* in which position of the se-

quence of Phases it is), the index of the previous and next Phases, the time-step length (dt), the name of the time unit (*e.g.,* second, minute), how long the Phase should be (τ, "`phase_duration`" in the code, see Section 5.2.2.1 and C.1.1.4), a flag setting the transition to the next Phase to be stochastic or deterministic (see Section 5.2.2.1 and C.1.1.4), a flag for mitosis or meiosis at phase exit, a flag for removal from the simulation at phase exit (*i.e.*, the cell dies, is killed, leaves the simulated domain). The Phase class also keeps track of the amount of time the cell has spent in the current Phase (T), and (optionally) the volume of the cell in the simulation. Although the cell volume definition and update is handled by the Cell Volume class (Section 5.2.1), the volume change rates is an attribute of the Phase class. It's important to note that the volume of the cell defined by PhenoCellPy's volume class may differ from the volume of the cell in the simulation.

The Phase class also has several functions defined, to time-step the model (Section C.1.1.2), to update the cellular volume model (Section 5.2.1), to double the target volume of the cellular volume model, and to evaluate if the transition to the next Phase should occur (Section 5.2.2.1 and C.1.1.4). It also has place-holder functions that can be replaced by user-defined ones, one that should be executed upon Phase entry (`entry_function(*args)`), one just before Phase exit (`exit_function(*args)`), one for exiting the Phenotype and entering senescence (`arrest_function(*args)`), and one that is executed at each time-step (`user_phase_time_step(*args)`), see Section C.1.2.4.

C.1.1.1 Phase class initialization

The Phase `__init__` function performs some checks on attribute values, *e.g.* $dt > 0$, a negative or zero time-step makes no sense, the custom functions should be functions, and so on. Initializes attributes to set values, and initializes the Cellular Volume model class. The Phase `__init__` function is in Supplemental Materials C.2.

C.1.1.2 Phase time-step

The Phase's time-step function is responsible for incrementing T (total time the cell's spent in the current Phase, `time_in_phase` in the code) by dt. Then it calls the volume update function and checks if the cell exits the Phenotype and goes into senescence. Finally, it calls the Phase transition function (the default deterministic or stochastic, or a user-defined function). It returns a tuple of two boolean flags, the first flags if the cell should go to the next Phase in the Phenotype, the second if the cell should exit the Phenotype and enter senescence. The Phase time-step function is shown in Listing 39.

C.1.1.3 Phases volume update

The volume update itself is handled by the Cell Volume class. As the volume change rates are an attribute of the Phase class, however, the Phase class has its own intermediary update volume function. This intermediary function's job is to pass the rates to the Cell Volume's update volume function. The intermediary function is shown in Listing 40.

C.1.1.4 Phase Transition

The transition from one phase to the next can be either deterministic (with a set period) or stochastic (with a set transition rate) by setting the Phase's class attribute ”`fixed_duration`” to be True (deterministic) or False (stochastic). Based on the flag, the Phase class sets its transition check function to be either `_transition_to_next_phase_deterministic` (for the deterministic case), Listing 41, or `_transition_to_next_phase_stochastic` (for the stochastic case), Listing 42. By default, `fixed_duration` is `False`, *i.e.*, the default behavior is to use the stochastic transition. A user can also define their own transition function that can take into account other factors. As the Phase transition function can be defined by the user, our default transition functions must also have `*args` as its argument.

```
def time_step_phenotype(self):
        self.time_in_phase += self.dt
        self.update_volume()
        transition_to_index = None
        if self.user_phase_time_step is not None:
            self.user_phase_time_step(*self.user_phase_time_step_args)
        if self.arrest_function is not None:
            exit_phenotype = self.arrest_function(
            *self.exit_function_args)
            go_to_next_phase_in_phenotype = False
            return go_to_next_phase_in_phenotype, exit_phenotype, \
                transition_to_index
        else:
            exit_phenotype = False
        go_to_next_phase_in_phenotype = \
            self.check_transition_to_next_phase_function(
                *self.check_transition_to_next_phase_function_args)
        if hasattr(go_to_next_phase_in_phenotype, "len") and \
            len(go_to_next_phase_in_phenotype) > 1:
                transition_to_index = go_to_next_phase_in_phenotype[1]
                go_to_next_phase_in_phenotype = \
                    go_to_next_phase_in_phenotype[0]
        if go_to_next_phase_in_phenotype and self.exit_function is not \
         None:
            self.exit_function(*self.exit_function_args)
            return go_to_next_phase_in_phenotype, exit_phenotype, \
                transition_to_index
        return go_to_next_phase_in_phenotype, exit_phenotype, \
            transition_to_index
```

Listing 39: Phase class time-step function.

```
def update_volume(self):
    self.volume.update_volume(self.dt, self.fluid_change_rate,
        self.nuclear_volume_change_rate,
        self.cytoplasm_volume_change_rate,
        self.calcification_rate)
```

Listing 40: Phase class volume update intermediary function.

```
def _transition_to_next_phase_deterministic(self, *none):
    return self.time_in_phase > self.phase_duration
```

Listing 41: Deterministic transition function

```
def _transition_to_next_phase_stochastic(self, *none):
    prob = 1 - exp(-self.dt / self.phase_duration)
    return uniform() < prob
```

Listing 42: Stochastic transition function

C.1.2 Cellular Phenotype

The Phenotype class has as attributes a descriptive name (*e.g.*, Standard necrosis model, Flow Cytometry Basic), the time-step length (dt), the name of the time unit (*e.g.,* second, minute), a list of Phases that make the Phenotype, the index of the Starting Phase, an optional "stand-alone" senescent Phase (i.e., a Phase that is outside the Phenotype cycle), the current Phase, and the amount of time spent in this Phenotype. Note that some of these attributes are shared with the Phase class, the Phenotype class should pass these shared attributes to its component Phases upon initialization.

This class has methods to time-step the phenotype model (Section C.1.2.2), to perform user-defined time-step tasks, to change the phenotype phase to an arbitrary phase of the phenotype cycle (Section C.1.2.3), to go to the next phase in the cycle (Section C.1.2.3), and to go to a non-changing senescent phase (Section C.1.2.4).

C.1.2.1 Phenotype class initialization

The Phenotype `__init__` function performs sanity checks and initializes attributes. Initialization of component Phases should be made in the Phenotype `__init__` function. The Phenotype class `__init__` function is shown in Listing 43.

```
def __init__(self, name: str = "unnamed", dt: float = 1,
time_unit: str = "min",
    phases: list = None, senescent_phase: Phases.Phase or False = None,
    starting_phase_index: int = 0):
    self.name = name
    self.time_unit = time_unit
    if dt <= 0 or dt is None:
        raise ValueError(f"'dt' must be greater than 0. Got {dt}.")
    self.dt = dt
    if phases is None:
        self.phases = [Phases.Phase(previous_phase_index=0,
            next_phase_index=0, dt=self.dt, time_unit=time_unit)]
    else:
        self.phases = phases
    if senescent_phase is None:
        self.senescent_phase = Phases.SenescentPhase(dt=self.dt)
    elif senescent_phase is not None and not senescent_phase:
        self.senescent_phase = False
    elif not isinstance(senescent_phase, Phases.Phase):
        raise ValueError(
            f"`senescent_phase` must Phases.Phase object, False, or"
            f" None."
            f" Got {senescent_phase}")
    else:
        self.senescent_phase = senescent_phase
    if starting_phase_index is None:
        starting_phase_index = 0
    self.current_phase = self.phases[starting_phase_index]
    self.time_in_phenotype = 0
```

Listing 43: Phenotype class `__init__` function

C.1.2.2 Phenotype time-step

The first thing the Phenotype time-step function (Listing 44) does is check if the Phenotype has just started (i.e., the time spent thus far in the Phenotype is 0), and if so it calls the initial Phase entry function. Then it increments the "amount of time spent in this Phenotype" attribute (`time_in_cycle` in the code) by dt, calls the current Phase's time-step function (Section C.1.1.2, and checks if the Phenotype should move to the next Phase or go to

quiescence (as determined by the Phase's time-step).

If the Phenotype should go to the next Phase it calls the `go_to_next_phase` function (Section C.1.2.3), if it should go to quiescence it calls the `go_to_quiescence` function (Section C.1.2.4). The time-step function returns a tuple of three boolean flags, the values of which can be determined by the `go_to_next_phase` function. The first flag of the tuple says if the Phenotype has changed Phases, the second if the simulated cell should be removed from the simulation, and the third if the simulated cell has undergone cell division.

C.1.2.3 Switching Phases

The `set_phase` function (Listing 45) is responsible for switching the Phase of a Phenotype, it does so by setting the `current_phase` to be the phase of index i. It also ensures that the Cell Volume sub-class attributes are correct in the case the cell has changed volume while in the current Phase and resets T (the time spent in the Phase) to zero. Finally, it calls the new Phase entry function if there is one.

The `go_to_next_phase` (Listing 46) switches to the next Phase by calling `set_phase` with the current Phase's "next phase index" attribute. Before switching Phases, it fetches the boolean flags for division and removal (*e.g.*, death, leaving the simulation domain) at Phase exit. It returns a tuple of three boolean flags: Phase change, cell removal from the simulation, and cell division.

```
def go_to_next_phase(self):
    changed_phases = True
    divides = self.current_phase.division_at_phase_exit
    removal = self.current_phase.removal_at_phase_exit
    self.set_phase(self.current_phase.next_phase_index)
    return changed_phases, removal, divides
```

Listing 46: Phenotype Class `go_to_next_phase` function

```
def time_step_phenotype(self):
            if not self.time_in_phenotype and \
                self.current_phase.entry_function is not None:
                    self.current_phase.entry_function(
                        *self.current_phase.entry_function_args)
        if self.user_phenotype_time_step is not None:
            self.user_phenotype_time_step(
                *self.user_pheno_time_step_args)
        self.time_in_phenotype += self.dt
        go_next_phase, exit_phenotype, transition_to_index = \
            self.current_phase.time_step_phase()
        if go_next_phase:
            if transition_to_index is not None:
                phase_idx = self.current_phase.index
                old_next_phase_idx = \
                    self.current_phase.next_phase_index
                self.current_phase.next_phase_index = \
                    transition_to_index
            else:
                phase_idx = None
            changed_phases, cell_removed, cell_divides = \
                self.go_to_next_phase()
            if phase_idx is not None:
                self.phases[phase_idx].next_phase_index  = \
                    old_next_phase_idx
            return changed_phases, cell_removed, cell_divides
        elif exit_phenotype:
            self.go_to_senescence()
            changed_phases, cell_removed, cell_divides = (True, False,
                False)
            return changed_phases, cell_removed, cell_divides
        changed_phases, cell_removed, cell_divides = (False, False,
            False)
        return changed_phases, cell_removed, cell_divides
```

Listing 44: Phenotype time-step function

```
def set_phase(self, idx):
    # get the current cytoplasm, nuclear, calcified volumes
    cyto_solid = self.current_phase.volume.cytoplasm_solid
    cyto_fluid = self.current_phase.volume.cytoplasm_fluid
    nucl_solid = self.current_phase.volume.nuclear_solid
    nucl_fluid = self.current_phase.volume.nuclear_fluid
    calc_frac = self.current_phase.volume.calcified_fraction
    # get the target volumes
    target_cytoplasm_solid = \
        self.current_phase.volume.cytoplasm_solid_target
    target_nuclear_solid = \
        self.current_phase.volume.nuclear_solid_target
    target_fluid_fraction = \
        self.current_phase.volume.target_fluid_fraction
    # set parameters of next phase
    self.phases[idx].volume.cytoplasm_solid = cyto_solid
    self.phases[idx].volume.cytoplasm_fluid = cyto_fluid
    self.phases[idx].volume.nuclear_solid = nucl_solid
    self.phases[idx].volume.nuclear_fluid = nucl_fluid
    self.phases[idx].volume.calcified_fraction = calc_frac
    self.phases[idx].volume.cytoplasm_solid_target = \
        target_cytoplasm_solid
    self.phases[idx].volume.nuclear_solid_target = \
        target_nuclear_solid
    self.phases[idx].volume.target_fluid_fraction = \
        target_fluid_fraction
    # set phase
    self.current_phase = self.phases[idx]
    self.current_phase.time_in_phase = 0
    if self.current_phase.entry_function is not None:
        self.current_phase.entry_function(
            *self.current_phase.entry_function_args)
```

Listing 45: Phenotype Class `set_phase` function.

C.1.2.4 Leaving the Phenotype

To exit the Phenotype the modeler has to define the optional arrest function which is a member of the Phase class (Section 5.2.2). The arrest function is called by the Phase time-step function (Section C.1.1.2) and its return value is used by the Phenotype time-step function to exit the Phenotype cycle.

If the arrest function returns `True`, the Phenotype time-step calls `go_to_senescence`. `go_to_senescence` checks that the Phenotype attribute `senescent_phase` is a class of type Phase, if that's the case it sets the current phase to be the special senescent Phase. If that's not the case the function will simply return early.

Before the Phase change, `go_to_quiescence` saves the volume parameters to temporary variables to keep them as they are after the change. As this is a change to a senescent Phase, all target volumes of the Cell Volume sub-class are set to be the current actual volumes (see Section 5.2.1 for definition of target volume and actual volumes). It resets `time_in_phase` to be 0. The `go_to_quiescence` function is shown in Listing 47.

```
def go_to_senescence(self):
    if not isinstance(self.senescent_phase, Phases.Phase):
        return
    # get the current cytoplasm, nuclear, calcified volumes
    cyto_solid = self.current_phase.volume.cytoplasm_solid
    cyto_fluid = self.current_phase.volume.cytoplasm_fluid
    nucl_solid = self.current_phase.volume.nuclear_solid
    nucl_fluid = self.current_phase.volume.nuclear_fluid
    calc_frac = self.current_phase.volume.calcified_fraction
    # setting the senescent phase volume parameters.
    # As the cell is now senescent it shouldn't want to change its
    # volume, so we set the targets to be the current measurements
    self.senescent_phase.volume.cytoplasm_solid = cyto_solid
    self.senescent_phase.volume.cytoplasm_fluid = cyto_fluid
    self.senescent_phase.volume.nuclear_solid = nucl_solid
    self.senescent_phase.volume.nuclear_fluid = nucl_fluid
    self.senescent_phase.volume.nuclear_solid_target = nucl_solid
    self.senescent_phase.volume.cytoplasm_solid_target = cyto_solid
    self.senescent_phase.volume.calcified_fraction = calc_frac
    self.senescent_phase.volume.target_fluid_fraction = \
        (cyto_fluid + nucl_fluid) / \
            (nucl_solid + nucl_fluid + cyto_fluid + cyto_solid)
    # set the phase to be senescent
    self.current_phase = self.senescent_phase
    self.current_phase.time_in_phase = 0
```

Listing 47: Phenotype Class `go_to_quiescence` function.

C.1.3 Cellular Volume

The update volume function is shown in Listing 48. It takes as arguments, in order, the time-step length (dt), r_F, r_{NS}, r_{CS}, and r_C. For stability reasons, we use SciPy's `odeint` function [75] to solve the volume dynamics, together with an intermediary function `volume_relaxation` (see Listing 49).

C.1.3.1 Cellular Volume defaults and init function

The Cellular Volume `__init__` function checks if the user defined custom values, checks that they are sensible (e.g., that there are no negative volume, or fractional volumes $\notin$

```
def update_volume(self, dt, fluid_change_rate,
    nuclear_volume_change_rate,
    cytoplasm_volume_change_rate, calcification_rate):
        dt_array = array([0, dt])
        self.fluid = odeint(self.volume_relaxation, self.fluid,
        dt_array,
                        args=(fluid_change_rate,
                            self.target_fluid_fraction * self.total))\
                            [-1][0]
        self.nuclear_fluid = (self.nuclear / (self.total + 1e-12)) * \
                                self.fluid
        self.cytoplasm_fluid = self.fluid - self.nuclear_fluid
        self.nuclear_solid = odeint(self.volume_relaxation,
                                self.nuclear_solid,
                                dt_array,
                                args=(nuclear_volume_change_rate,
                                      self.nuclear_solid_target))[-1][0]
        self.cytoplasm_solid_target = \
            self.target_cytoplasm_to_nuclear_ratio * \
                self.nuclear_solid_target
        self.cytoplasm_solid = odeint(
                                self.volume_relaxation,
                                self.cytoplasm_solid, dt_array,
                                args=(cytoplasm_volume_change_rate,
                                    self.cytoplasm_solid_target))[-1][0]
        self.solid = self.nuclear_solid + self.cytoplasm_solid
        self.nuclear = self.nuclear_solid + self.nuclear_fluid
        self.cytoplasm = self.cytoplasm_fluid + self.cytoplasm_solid
        self.calcified_fraction = odeint(
                                    self.volume_relaxation,
                                    self.calcified_fraction, dt_array,
                                    args=(calcification_rate, 1))[-1][0]
        self.total = self.cytoplasm + self.nuclear
        self.fluid_fraction = self.fluid / (self.total + 1e-12)
```

Listing 48: Cell Volume class `update_volume` function.

```
@staticmethod
def volume_relaxation(current_volume, t, rate, target_volume):
    dvdt = rate * (target_volume - current_volume)
    return dvdt
```

Listing 49: Intermediary `volume_relaxation` function.

$[0, 1]$), and initializes the class' attributes.

C.2 Phase init function

```
def __init__(self, index: int = None, previous_phase_index: int = None,
    next_phase_index: int = None, dt: float = None,
    time_unit: str = "min", name: str = None,
    division_at_phase_exit: bool = False,
    removal_at_phase_exit: bool = False, fixed_duration: bool = False,
    phase_duration: float = 10, entry_function=None,
    entry_function_args: list = None,
    exit_function=None, exit_function_args: list = None,
    arrest_function=None,
    arrest_function_args: list = None, transition_to_next_phase=None,
    transition_to_next_phase_args: list = None,
    simulated_cell_volume: float = None,
    cytoplasm_volume_change_rate=None,
    nuclear_volume_change_rate=None, calcification_rate=None,
    target_fluid_fraction=None, nuclear_fluid=None, nuclear_solid=None,
    nuclear_solid_target=None, cytoplasm_fluid=None,
    cytoplasm_solid=None,
    cytoplasm_solid_target=None, target_cytoplasm_to_nuclear_ratio=None,
    calcified_fraction=None, fluid_change_rate=None,
    relative_rupture_volume=None):

    if index is None:
        self.index = 0
    else:
        self.index = index
    self.previous_phase_index = previous_phase_index
    self.next_phase_index = next_phase_index
    self.time_unit = time_unit
    if dt is None or dt <= 0:
        raise ValueError(f"'dt' must be greater than 0. Got {dt}.")
    self.dt = dt
```

```
if name is None:
    self.name = "unnamed"
else:
    self.name = name
self.division_at_phase_exit = division_at_phase_exit
self.removal_at_phase_exit = removal_at_phase_exit
self.fixed_duration = fixed_duration
if phase_duration <= 0:
    raise ValueError(f"'phase_duration' must be greater than 0."
                     f" Got {phase_duration}")
self.phase_duration = phase_duration
self.time_in_phase = 0
self.entry_function = entry_function
self.entry_function_args = entry_function_args
self.exit_function = exit_function
self.exit_function_args = exit_function_args
if self.exit_function is not None and \
    not (type(self.exit_function_args) == list or
         type(self.exit_function_args) == tuple):
    raise TypeError("Exit function defined but no args given. "
                    f"Was expecting "
                    f"'exit_function_args' to be a list or tupple,"
                    f" got {type(exit_function_args)}.")

self.arrest_function = arrest_function
self.arrest_function_args = arrest_function_args
if self.arrest_function is not None and \
  type(self.arrest_function_args) != list:
    raise TypeError("Arrest function defined but no args given."
                    "Was expecting "
                    f"'arrest_function_args' to be a list,"
                    f"got {type(arrest_function_args)}.")
```

```
        if transition_to_next_phase is None:
            self.transition_to_next_phase_args = [None]
            if fixed_duration:
                self.transition_to_next_phase = \
                    self._transition_to_next_phase_deterministic
            else:
                self.transition_to_next_phase = \
                    self._transition_to_next_phase_stochastic
        else:
            if type(transition_to_next_phase_args) != list:
                raise TypeError("Custom exit function selected but no args "
                                "given. Was expecting "
                                "'transition_to_next_phase_args' to be a"
                                " list, got "
                                f"{type(transition_to_next_phase_args)}.")
            self.transition_to_next_phase_args = \
                transition_to_next_phase_args
            self.transition_to_next_phase = transition_to_next_phase
        if simulated_cell_volume is None:
            self.simulated_cell_volume = 1
        else:
            self.simulated_cell_volume = simulated_cell_volume

        # the default rates are reference values for MCF-7, in 1/min
        if cytoplasm_volume_change_rate is None:
            self.cytoplasm_volume_change_rate = 0.27 / 60.0
        else:
            self.cytoplasm_volume_change_rate = cytoplasm_volume_change_rate
        if nuclear_volume_change_rate is None:
            self.nuclear_volume_change_rate = 0.33 / 60.0
        else:
            self.nuclear_volume_change_rate = nuclear_volume_change_rate
        if calcification_rate is None:
```

```
        self.calcification_rate = 0
    else:
        if calcification_rate < 0:
            raise ValueError(f"`calcification_rate` must be >= 0,"
                             f" got {calcification_rate}")
        self.calcification_rate = calcification_rate
    if fluid_change_rate is None:
        self.fluid_change_rate = 3.0 / 60.0
    else:
        self.fluid_change_rate = fluid_change_rate
    self.volume = \
        CellVolumes(
            target_fluid_fraction=target_fluid_fraction,
            nuclear_fluid=nuclear_fluid,
            nuclear_solid=nuclear_solid,
            nuclear_solid_target=nuclear_solid_target,
            cytoplasm_fluid=cytoplasm_fluid,
            cytoplasm_solid=cytoplasm_solid,
            cytoplasm_solid_target=cytoplasm_solid_target,
            target_cytoplasm_to_nuclear_ratio=
                target_cytoplasm_to_nuclear_ratio,
            calcified_fraction=calcified_fraction,
            relative_rupture_volume=relative_rupture_volume)
```

C.3 Phase init function

```
def __init__(self, target_fluid_fraction=None, nuclear_fluid=None,
    nuclear_solid=None,
    nuclear_solid_target=None,cytoplasm_fluid=None,
    cytoplasm_solid=None,
    cytoplasm_solid_target=None,
    target_cytoplasm_to_nuclear_ratio=None,
    calcified_fraction=None, relative_rupture_volume=None):
    _total = 2494
    _fluid_fraction = .75
    _fluid = _fluid_fraction * _total
    _solid = _total - _fluid
    _nuclear = 540
    _nuclear_fluid = _fluid_fraction * _nuclear
    _nuclear_solid = _nuclear - _nuclear_fluid
    _cytoplasm = _total - _nuclear
    _cytoplasm_fluid = _fluid_fraction * _cytoplasm
    _cytoplasm_solid = _cytoplasm - _cytoplasm_fluid
    _calcified_fraction = 0
    _relative_rupture_volume = 100
    if target_fluid_fraction is None:
        self.target_fluid_fraction = _fluid_fraction
    else:
        if not 0 <= target_fluid_fraction <= 1:
            raise ValueError(f"`target_fluid_fraction` must be in "
                f"range [0, 1]. Got {target_fluid_fraction}")
        self.target_fluid_fraction = target_fluid_fraction
    if nuclear_fluid is None:
        self.nuclear_fluid = _nuclear * self.target_fluid_fraction
    else:
        if nuclear_fluid < 0:
            raise ValueError(f"`nuclear_fluid` must be >=0."
```

```
            f" Got {nuclear_fluid}")
        self.nuclear_fluid = nuclear_fluid
    if nuclear_solid is None:
        self.nuclear_solid = _nuclear * (1 - self.target_fluid_fraction)
    else:
        if nuclear_solid < 0:
            raise ValueError(f"`nuclear_solid` must be >=0."
                f" Got {nuclear_solid}")
        self.nuclear_solid = nuclear_solid
    if nuclear_solid_target is None:
        self.nuclear_solid_target = self.nuclear_solid
    else:
        if nuclear_solid_target < 0:
            raise ValueError(f"`nuclear_solid_target` must be >=0."
                " Got {nuclear_solid_target}")
        self.nuclear_solid_target = nuclear_solid_target
    if cytoplasm_fluid is None:
        self.cytoplasm_fluid = _cytoplasm * self.target_fluid_fraction
    else:
        self.cytoplasm_fluid = cytoplasm_fluid
    if cytoplasm_solid is None:
        self.cytoplasm_solid = _cytoplasm * (1 - self.
                                             target_fluid_fraction)
    else:
        self.cytoplasm_solid = cytoplasm_solid
    if cytoplasm_solid_target is None:
        self.cytoplasm_solid_target = self.cytoplasm_solid
    else:
        self.cytoplasm_solid_target = cytoplasm_solid_target
    self.cytoplasm = self.cytoplasm_fluid + self.cytoplasm_solid
    self.nuclear = self.nuclear_fluid + self.nuclear_solid
    if target_cytoplasm_to_nuclear_ratio is None:
        self.target_cytoplasm_to_nuclear_ratio = \
```

```
            self.cytoplasm / (1e-16 + self.nuclear)
    else:
        self.target_cytoplasm_to_nuclear_ratio = \
            target_cytoplasm_to_nuclear_ratio
    if calcified_fraction is None:
        self.calcified_fraction = _calcified_fraction
    else:
        self.calcified_fraction = calcified_fraction
    if relative_rupture_volume is None:
        self.relative_rupture_volume = _relative_rupture_volume
    else:
        self.relative_rupture_volume = relative_rupture_volume
    self.fluid = self.cytoplasm_fluid + self.nuclear_fluid
    self.solid = self.cytoplasm_solid + self.nuclear_solid
    self.total = self.nuclear + self.cytoplasm
    self.fluid_fraction = self.fluid / self.total
    self.rupture_volume = self.relative_rupture_volume * self.total
```

C.4 Ki-67 Basic Cycle Improved Division Implementation in CompuCell3D

Note: we have removed the implementation of population statistics, data saving, and plots from the code presented here. In places this would cause an error we have added a `pass` statement. The model that comes packaged with PhenoCellPy does include population statistics, data saving, and plots.

```
from cc3d.cpp.PlayerPython import *
from cc3d import CompuCellSetup
from cc3d.core.PySteppables import *
from numpy import median, quantile, nan
import sys
sys.path.extend(["C:\\PhenoCellPy"])
import Phenotypes as pheno
def Ki67pos_transition(*args):
    # print(len(args), print(args))
    # args = [cc3d cell volume, phase's target volume, time in phase,
                                        phase duration
    return args[0] >= args[1] and args[2] > args[3]

class ConstraintInitializerSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)
        self.track_cell_level_scalar_attribute(field_name='
                                               phase_index_plus_1',
                                               attribute_name='
                                               phase_index_plus_1')
        self.target_volume = None
        self.doubling_volume = None
        self.volume_conversion_unit = None

    def start(self):
        side = 10
```

```
        self.target_volume = side * side
        self.doubling_volume = 2 * self.target_volume
        x = self.dim.x // 2 - side // 2
        y = self.dim.x // 2 - side // 2
        cell = self.new_cell(self.CELL)
        self.cell_field[x:x + side, y:y + side, 0] = cell
        dt = 5  # 5 min/mcs
        ki67_basic_modified_transition = \
            pheno.phenotypes.Ki67Basic(dt=dt,
                transitions_to_next_phase=[None,
                    Ki67pos_transition],
                transitions_to_next_phase_args=[None,
                    [-9, 1, -9, 1]])
        self.volume_conversion_unit = self.target_volume / \
            ki67_basic_modified_transition.current_phase.volume.total
        for cell in self.cell_list:
            cell.targetVolume = self.target_volume
            cell.lambdaVolume = 2.0
            pheno.utils.add_phenotype_to_CC3D_cell(cell,
                ki67_basic_modified_transition)
            cell.dict["phase_index_plus_1"] = \
                cell.dict["phenotype"].current_phase.index + 1
        self.shared_steppable_vars["constraints"] = self


class MitosisSteppable(MitosisSteppableBase):
    def __init__(self, frequency=1):
        MitosisSteppableBase.__init__(self, frequency)
        self.constraint_vars = None
        self.previous_number_cells = 0
        self.plot = True
        self.save = False
        if self.save:
            pass
```

```
    def start(self):
        self.constraint_vars = self.shared_steppable_vars["constraints"]
        if self.plot:
            pass
    def step(self, mcs):
        if not mcs and self.plot:
            pass
        elif not mcs % 50 and \
            len(self.cell_list) - self.previous_number_cells > 0 \
            and self.plot:
            pass
        cells_to_divide = []
        n_zero = 0
        n_one = 0
        volumes = []
        time_spent_in_0 = []
        time_spent_in_1 = []
        for cell in self.cell_list:
            volumes.append(cell.volume)
            cell.dict["phenotype"].current_phase.simulated_cell_volume =
                                                cell.volume
            if cell.dict["phenotype"].current_phase.index == 0:
                n_zero += 1
                time_spent_in_0.append(cell.dict["phenotype"].
                                                    current_phase.
                                                    time_in_phase)
            elif cell.dict["phenotype"].current_phase.index == 1:
                n_one += 1
                time_spent_in_1.append(cell.dict["phenotype"].
                                                    current_phase.
                                                    time_in_phase)
                # args = [cc3d cell volume,
                #         doubling colume,
```

```
            #         time in phase,
            #         phase duration]
            args = [
                cell.volume,
                .9 * self.constraint_vars.doubling_volume,
                # we use 90\% of the doubling volume because cc3d
                                                    cells
                # will always be slightly below their target due to
                                                    the contact
                                                    energy
                cell.dict["phenotype"].current_phase.time_in_phase \
                    + cell.dict["phenotype"].dt,
                cell.dict["phenotype"].current_phase.phase_duration]
            cell.dict["phenotype"].current_phase.
                transition_to_next_phase_args = \
                args
        changed_phase, should_be_removed, divides = \
            cell.dict["phenotype"].time_step_phenotype()
        converted_volume = \
            self.constraint_vars.volume_conversion_unit * \
                cell.dict["phenotype"].current_phase.volume.total
        cell.targetVolume = converted_volume
        if changed_phase:
            cell.dict["phase_index_plus_1"] = \
                cell.dict["phenotype"].current_phase.index + 1
        if divides:
            cells_to_divide.append(cell)
    if self.save or self.plot:
        pass
    for cell in cells_to_divide:
        # self.divide_cell_random_orientation(cell)
        # Other valid options
        # self.divide_cell_orientation_vector_based(cell,1,1,0)
```

```
            # self.divide_cell_along_major_axis(cell)
            self.divide_cell_along_minor_axis(cell)

    def update_attributes(self):
        # resetting target volume
        converted_volume = \
            self.constraint_vars.volume_conversion_unit * \
                self.parent_cell.dict["phenotype"].current_phase.volume.
                                                        total
        self.parent_cell.targetVolume = converted_volume
        self.clone_parent_2_child()
        self.parent_cell.dict["phase_index_plus_1"] = \
            self.parent_cell.dict["phenotype"].current_phase.index + 1
        self.child_cell.dict["phase_index_plus_1"] = \
            self.child_cell.dict["phenotype"].current_phase.index + 1
        self.child_cell.dict["phenotype"].time_in_phenotype = 0
    def on_stop(self):
        self.finish()
    def finish(self):
        if self.save:
            pass
```

C.5 Ki-67 Basic Cycle Implementation in Tissue Forge

```
import tissue_forge as tf
import numpy as np
import sys
sys.path.extend(["C:\\PhenoCellPy"])
import Phenotypes as pheno
def get_radius_sphere(volume):
    return ((1 / (np.pi * 4 / 3)) * volume) ** (1 / 3)
# potential cutoff distance
cutoff = 3
# space set up
dim = [50, 50, 50]
tf.init(dim=dim, cutoff=cutoff)
pot = tf.Potential.morse(d=3, a=5, min=-0.8, max=2)
# Particle types
mass = 40
radius = .4
global density
density = mass / ((4 / 3) * np.pi * radius * radius * radius)
dt = 10  # min/time step
ki67_basic = pheno.phenotypes.Ki67Basic(dt=dt)
global volume_conversion_unit
volume_conversion_unit = mass/ki67_basic.current_phase.volume.total
class CellType(tf.ParticleTypeSpec):
    mass = mass
    target_temperature = 0
    radius = radius
    dynamics = tf.Overdamped
    cycle = ki67_basic
Cell = CellType.get()
tf.bind.types(pot, Cell, Cell)
rforce = tf.Force.random(mean=0, std=50)
```

```
# bind it just like any other force
tf.bind.force(rforce, Cell)
first_cell = Cell([d // 2 for d in dim])
first_cell.cycle = ki67_basic
global cells_cycles
cells_cycles = {f"{first_cell.id}": ki67_basic}
def step_cycle_and_divide(event):
    for p in Cell.items():
        pcycle = cells_cycles[f"{p.id}"]
        pcycle.current_phase.simulated_cell_volume = p.mass * density
        phase_change, should_be_removed, division = pcycle.
                                                time_step_phenotype()
        radius = get_radius_sphere(
                volume_conversion_unit*pcycle.current_phase.volume.total
                                                        )
        # book-keeping, making sure the simulated cell grows
        p.radius = radius
        p.mass = ((4 / 3) * np.pi * radius * radius * radius) * density
        # if division occurs, divide
        if division:
            print("@@@\nDIVISION\n@@@")
            # save cell attribs to halve later
            cur_mass = p.mass
            # divide and reasign attribs (is this step necessary?)
            child = p.split()
            cells_cycles[f"{child.id}"] = ki67_basic
            child.mass = p.mass = cur_mass / 2
            child.radius = p.radius = get_radius_sphere((cur_mass / 2) /
                                                    density)
            cells_cycles[f"{child.id}"].volume = child.mass * density
            cells_cycles[f"{child.id}"].simulated_cell_volume = child.
                                                    mass * density
    return 0
```

```
tf.event.on_time(invoke_method=step_cycle_and_divide, period=.9*tf.
                                        Universe.dt)
# run the simulator interactive
tf.run()
```

REFERENCES

[1] T. J. Sego et al. “A modular framework for multiscale, multicellular, spatiotemporal modeling of acute primary viral infection and immune response in epithelial tissues and its application to drug therapy timing and effectiveness”. In: *PLOS Computational Biology* 16.12 (Dec. 2020), e1008451.

[2] Juliano Ferrari Gianlupi et al. “Multiscale Model of Antiviral Timing, Potency, and Heterogeneity Effects on an Epithelial Tissue Patch Infected by SARS-CoV-2”. In: *Viruses* 14.3 (Mar. 2022), p. 605.

[3] T. J. Sego et al. “Generation of multicellular spatiotemporal models of population dynamics from ordinary differential equations, with applications in viral infection”. In: *BMC Biology* 19.1 (Sept. 2021), p. 196.

[4] TJ Sego et al. “A multiscale multicellular spatiotemporal model of local influenza infection and immune response”. In: *Journal of Theoretical Biology* 532 (2022), p. 110918.

[5] Vivi Andasari et al. “Integrating Intracellular Dynamics Using CompuCell3D and Bionetsolver: Applications to Multiscale Modelling of Cancer Cell Growth and Invasion”. In: *PLOS ONE* 7.3 (Mar. 2012), pp. 1–17.

[6] Jonathan Ozik et al. “High-throughput cancer hypothesis testing with an integrated PhysiCell-EMEWS workflow”. In: *BMC Bioinformatics* 19.18 (Dec. 2018), p. 483.

[7] François Graner and James A Glazier. “Simulation of biological cell sorting using a two-dimensional extended Potts model”. In: *Physical review letters* 69.13 (1992), p. 2013.

[8] Susan D Hester et al. “A multi-cell, multi-scale model of vertebrate segmentation and somite formation”. In: *PLoS computational biology* 7.10 (2011), e1002155.

[9] Priyom Adhyapok et al. “A mechanical model of early somite segmentation”. In: *Iscience* 24.4 (2021), p. 102317.

[10] Maciej H. Swat et al. “Multi-Scale Modeling of Tissues Using CompuCell3D”. In: *Methods in cell biology* 110 (2012), pp. 325–366.

[11] Ahmadreza Ghaffarizadeh et al. “PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems”. In: *PLoS Comput Biol* 14.2 (Feb. 2018), e1005991.

[12] M. Hucka et al. “The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models”. In: *Bioinformatics* 19.4 (Mar. 2003), pp. 524–531.

[13] Juliano Ferrari Gianlupi et al. “PhenoCellPy: A Python package for biological cell behavior modeling”. In: (Apr. 2023).

[14] Richard T. Eastman et al. “Remdesivir: A Review of Its Discovery and Development Leading to Emergency Use Authorization for Treatment of COVID-19”. In: *ACS Central Science* 6.5 (May 2020), pp. 672–683.

[15] Siyuan Hao et al. “Long-Term Modeling of SARS-CoV-2 Infection of In Vitro Cultured Polarized Human Airway Epithelium”. In: *MBio* 11.6 (2020), p. 17.

[16] Josua O. Aponte-Serrano et al. “Multicellular spatial model of RNA virus replication and interferon responses reveals factors controlling plaque growth dynamics”. In: *PLOS Computational Biology* 17.10 (Oct. 2021), e1008874.

[17] Arash Keshavarzi Arshadi et al. “Artificial Intelligence for COVID-19 Drug Discovery and Vaccine Development”. In: *Frontiers in Artificial Intelligence* 3 (2020), p. 65.

[18] Adrianne L. Jenner et al. “Leveraging Computational Modeling to Understand Infectious Diseases”. In: *Current Pathobiology Reports* 8.4 (Dec. 2020), pp. 149–161.

[19] Stanca M. Ciupe. “Modeling the dynamics of hepatitis B infection, immunity, and drug therapy”. In: *Immunological Reviews* 285.1 (2018), pp. 38–54.

[20] Katharine Best and Alan S. Perelson. “Mathematical modeling of within-host Zika virus dynamics”. In: *Immunological Reviews* 285.1 (2018), pp. 81–96.

[21] Joshua T. Schiffer et al. “Herpes simplex virus-2 dynamics as a probe to measure the extremely rapid and spatially localized tissue-resident T-cell response”. In: *Immunological Reviews* 285.1 (2018), pp. 113–133.

[22] Yu-chen Cao, Qi-xin Deng, and Shi-xue Dai. “Remdesivir for severe acute respiratory syndrome coronavirus 2 causing COVID-19: An evaluation of the evidence”. In: *Travel Medicine and Infectious Disease* 35 (May 2020), p. 101647.

[23] Rita Humeniuk et al. “Safety, Tolerability, and Pharmacokinetics of Remdesivir, An Antiviral for Treatment of COVID-19, in Healthy Subjects”. In: *Clinical and Translational Science* 13.5 (2020), pp. 896–906.

[24] Christoph D. Spinner et al. “Effect of Remdesivir vs Standard Care on Clinical Status at 11 Days in Patients With Moderate COVID-19: A Randomized Clinical Trial”. In: *JAMA* 324.11 (Sept. 2020), pp. 1048–1057.

[25] Goran Kokic et al. “Mechanism of SARS-CoV-2 polymerase stalling by remdesivir”. In: *Nature communications* 12.1 (2021), pp. 1–7.

[26] Oriol Mitjà and Bonaventura Clotet. “Use of antiviral drugs to reduce COVID-19 transmission”. In: *The Lancet Global Health* 8.5 (2020), e639–e640.

[27] Erik De Clercq and Guangdi Li. “Approved antiviral drugs over the past 50 years”. In: *Clinical microbiology reviews* 29.3 (2016), pp. 695–747.

[28] Carolin Zitzmann and Lars Kaderali. “Mathematical analysis of viral replication dynamics and antiviral treatment strategies: from basic models to age-based multi-scale modeling”. In: *Frontiers in microbiology* 9 (2018), p. 1546.

[29] Pengxing Cao and James M McCaw. “The mechanisms for within-host influenza virus control affect model-based assessment and prediction of antiviral treatment”. In: *Viruses* 9.8 (2017), p. 197.

[30] Kwang Su Kim et al. “Incomplete antiviral treatment may induce longer durations of viral shedding during SARS-CoV-2 infection”. In: *Life science alliance* 4.10 (2021).

[31] Hana M. Dobrovolny. “Quantifying the effect of remdesivir in rhesus macaques infected with SARS-CoV-2”. In: *Virology* 550 (Nov. 2020), pp. 61–69.

[32] Brandi N Williamson et al. “Clinical benefit of remdesivir in rhesus macaques infected with SARS-CoV-2”. In: *Nature* 585.7824 (2020), pp. 273–276.

[33] James M. Gallo. “Hybrid physiologically-based pharmacokinetic model for remdesivir: Application to SARS-CoV-2”. In: *Clinical and Translational Science* 14.3 (2021), pp. 1082–1091.

[34] Ashish Goyal, E. Fabian Cardozo-Ojeda, and Joshua T. Schiffer. “Potency and timing of antiviral therapy as determinants of duration of SARS-CoV-2 shedding and intensity of inflammatory response”. In: *Science Advances* (Oct. 2020), eabc7112.

[35] Ashish Goyal et al. “Mathematical modeling explains differential SARS CoV-2 kinetics in lung and nasal passages in remdesivir treated rhesus macaques”. In: *bioRxiv* (June 2020), p. 2020.06.21.163550.

[36] Veronika I Zarnitsyna et al. “Advancing therapies for viral infections using mechanistic computational models of the dynamic interplay between the virus and host immune response”. In: *Current opinion in virology* 50 (2021), pp. 103–109.

[37] James A Glazier, Ariel Balter, and Nikodem J Popławski. “Magnetization to morphogenesis: a brief history of the Glazier-Graner-Hogeweg model”. In: *Single-Cell-Based Models in Biology and Medicine*. Springer, 2007, pp. 79–106.

[38] Eric Bonabeau. “Agent-based modeling: Methods and techniques for simulating human systems”. In: *Proceedings of the national academy of sciences* 99.suppl 3 (2002), pp. 7280–7287.

[39] Abbas Shirinifard et al. “3D Multi-Cell Simulation of Tumor Growth and Angiogenesis”. In: *PLOS ONE* 4.10 (Oct. 2009), pp. 1–11.

[40] Thomas J Gast et al. “A computational model of peripheral photocoagulation for the prevention of progressive diabetic capillary occlusion”. In: *Journal of diabetes research* 2016 (2016).

[41] Michael Getz et al. *Iterative community-driven development of a SARS-CoV-2 tissue simulator*. Nov. 2021.

[42] Chase Cockrell and Gary An. “Comparative computational modeling of the bat and human immune response to viral infection with the Comparative Biology Immune Agent Based Model”. In: *Viruses* 13.8 (2021), p. 1620.

[43] Zhi Zeng et al. “Pulmonary pathology of early-phase COVID-19 pneumonia in a patient with a benign lung lesion”. In: *Histopathology* 77.5 (2020), pp. 823–831.

[44] Myriam Remmelink et al. “Unspecific post-mortem findings despite multiorgan viral spread in COVID-19 patients”. In: *Critical care* 24.1 (2020), pp. 1–10.

[45] Jessica K Fiege et al. “Single cell resolution of SARS-CoV-2 tropism, antiviral responses, and susceptibility to therapies in primary human airway epithelium”. In: *PLoS pathogens* 17.1 (2021), e1009292.

[46] Alvaro A Ordonez et al. “Dynamic imaging in patients with tuberculosis reveals heterogeneous drug exposures in pulmonary lesions”. In: *Nature medicine* 26.4 (2020), pp. 529–534.

[47] Shani Ben-Moshe and Shalev Itzkovitz. “Spatial heterogeneity in the mammalian liver”. In: *Nature reviews Gastroenterology & hepatology* 16.7 (2019), pp. 395–410.

[48] Juan Guillermo Diaz Ochoa et al. “A multi-scale modeling framework for individualized, spatiotemporal prediction of drug effects and toxicological risk”. In: *Frontiers in pharmacology* 3 (2013), p. 204.

[49] Keertan Dheda et al. “Drug-penetration gradients associated with acquired drug resistance in patients with tuberculosis”. In: *American journal of respiratory and critical care medicine* 198.9 (2018), pp. 1208–1219.

[50] Stefan Hoops et al. “COPASI—a COmplex PAthway SImulator”. In: *Bioinformatics* 22.24 (Dec. 2006), pp. 3067–3074.

[51] European Medicines Agency. “Summary on compassionate use. Remdesivir. EMA/178637/2020—Rev.2”. In: (Apr. 2020).

[52] Ka-Tim Choy et al. “Remdesivir, lopinavir, emetine, and homoharringtonine inhibit SARS-CoV-2 replication in vitro”. In: *Antiviral Research* 178 (June 2020), p. 104786.

[53] Andrés Pizzorno et al. “In vitro evaluation of antiviral activity of single and combined repurposable drugs against SARS-CoV-2”. In: *Antiviral Research* 181 (Sept. 2020), p. 104878.

[54] Patrick O. Hanafin et al. “A mechanism-based pharmacokinetic model of remdesivir leveraging interspecies scaling to simulate COVID-19 treatment in humans”. In: *CPT: Pharmacometrics & Systems Pharmacology* (Jan. 2021).

[55] Duxin Sun. “Remdesivir for treatment of COVID-19: combination of pulmonary and IV administration may offer aditional benefit”. In: *The AAPS journal* 22 (2020), pp. 1–6.

[56] Y. Y. Lew and T. I. Michalak. “In vitro and in vivo infectivity and pathogenicity of the lymphoid cell-derived woodchuck hepatitis virus”. In: *Journal of Virology* 75.4 (Feb. 2001), pp. 1770–1782.

[57] André Mateus et al. “Prediction of intracellular exposure bridges the gap between target- and cell-based drug discovery”. In: *Proceedings of the National Academy of Sciences* 114.30 (July 2017), E6231–E6239.

[58] Yanqun Wang et al. “Kinetics of viral load and antibody response in relation to COVID-19 severity”. In: *The Journal of clinical investigation* 130.10 (2020), pp. 5235 –5244.

[59] Roman Wölfel et al. “Virological assessment of hospitalized patients with COVID-2019”. In: *Nature* 581.7809 (2020), pp. 465–469.

[60] Vladimir Reinharz et al. “Understanding Hepatitis B Virus Dynamics and the Antiviral Effect of Interferon Alpha Treatment in Humanized Chimeric Mice”. In: *Journal of Virology* 95.14 (June 2021), e00492–20.

[61] Wen-juan Hu et al. “Pharmacokinetics and tissue distribution of remdesivir and its metabolites nucleotide monophosphate, nucleotide triphosphate, and nucleoside in mice”. In: *Acta Pharmacologica Sinica* 42.7 (2021), pp. 1195–1200.

[62] RECOVERY Collaborative Group. “Tocilizumab in patients admitted to hospital with COVID-19 (RECOVERY): a randomised, controlled, open-label, platform trial”. In: *The Lancet* 397.10285 (May 2021), pp. 1637–1645.

[63] Hui Chen et al. “Corticosteroid Therapy Is Associated With Improved Outcome in Critically Ill Patients With COVID-19 With Hyperinflammatory Phenotype”. In: *Chest* 159.5 (May 2021), pp. 1793–1802.

[64] Yeming Wang et al. “Remdesivir in adults with severe COVID-19: a randomised, double-blind, placebo-controlled, multicentre trial”. In: *The lancet* 395.10236 (2020), pp. 1569–1578.

[65] Mark D. Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Sci Data* 3.1 (Mar. 2016), p. 160018.

[66] Julio M. Belmonte et al. “Virtual-tissue computer simulations define the roles of cell adhesion and proliferation in the onset of kidney cystic disease”. In: *MBoC* 27.22 (Nov. 2016), pp. 3673–3685.

[67] Ahmadreza Ghaffarizadeh, Samuel H. Friedman, and Paul Macklin. “BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations”. In: *Bioinformatics* 32.8 (Apr. 2016), pp. 1256–1258.

[68] Nicholas J. Savill and Paulien Hogeweg. “Modelling Morphogenesis: From Single Cells to Crawling Slugs”. In: *Journal of Theoretical Biology* 184.3 (Feb. 1997), pp. 229–235.

[69] Renfrey Burnard Potts. “Some generalized order-disorder transformations”. In: *Mathematical proceedings of the cambridge philosophical society*. Vol. 48. Cambridge University Press. 1952, pp. 106–109.

[70] Elizabeth A Holm et al. “Effects of lattice anisotropy and temperature on domain growth in the two-dimensional Potts model”. In: *Physical Review A* 43.6 (1991), p. 2662.

[71] Nicholas Metropolis et al. “Equation of state calculations by fast computing machines”. In: *The journal of chemical physics* 21.6 (1953), pp. 1087–1092.

[72] W Keith Hastings. “Monte Carlo sampling methods using Markov chains and their applications”. In: (1970).

[73] P. Van Liedekerke et al. “Simulating tissue mechanics with agent-based models: concepts, perspectives and some novel results”. In: *Comp. Part. Mech.* 2.4 (Dec. 2015), pp. 401–444.

[74] Emanuel F. Teixeira, Heitor C. M. Fernandes, and Leonardo G. Brunnet. “A single active ring model with velocity self-alignment”. In: *Soft Matter* 17.24 (June 2021), pp. 5991–6000.

[75] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272.

[76] Harvey Lodish et al. *Molecular cell biology*. Macmillan, 2008.

[77] *CompuCell3D Reference Manual - 4.3.0 — CC3D Reference Manual 4.2.4 documentation*.

[78] András Szabó and Roeland MH Merks. “Cellular potts modeling of tumor growth, tumor invasion, and tumor evolution”. In: *Frontiers in oncology* 3 (2013), p. 87.

[79] *Python Scripting Manual for CompuCell3D - version 4.2.4*.

[80] Furkan;Sundus Kurtoglu. “Cycle Training App for PhysiCell”. In: (2020).

[81] Randy;Macklin Heiland. “PhysiCell biorobots simulation”. In: (2019).

[82] T.J. Sego et al. “Tissue Forge: Interactive Biological and Biophysics Simulation Environment”. In: *bioRxiv* (2022).

[83] Juliano Ferrari Gianlupi. *PhenoCellPy*. Dec. 2022.

[84] *MCF-7 - an overview | ScienceDirect Topics*.

[85] Lucian P. Smith et al. “Antimony: a modular model definition language”. In: *Bioinformatics* 25.18 (Sept. 2009), pp. 2452–2454.

[86] Gautier Stoll et al. “MaBoSS 2.0: an environment for stochastic Boolean modeling”. In: *Bioinformatics* 33.14 (July 2017), pp. 2226–2228.

[87] *MCF-10A Cell Line - an overview | ScienceDirect Topics*.

[88] S R McKeown. “Defining normoxia, physoxia and hypoxia in tumours—implications for treatment response”. In: *BJR* 87.1035 (Mar. 2014), p. 20130676.

[89] Geoffrey M. Cooper. “The Eukaryotic Cell Cycle”. In: *The Cell: A Molecular Approach. 2nd edition* (2000).

[90] Paul Macklin et al. “Patient-calibrated agent-based modelling of ductal carcinoma in situ (DCIS): From microscopic measurements to macroscopic predictions of clinical progression”. In: *Journal of Theoretical Biology* 301 (May 2012), pp. 122–140.

[91] Paul Macklin, Shannon Mumenthaler, and John Lowengrub. “Modeling Multiscale Necrotic and Calcified Tissue Biomechanics in Cancer Patients: Application to Ductal Carcinoma In Situ (DCIS)”. In: *Multiscale Computer Modeling in Biomechanics and Biomedical Engineering*. Ed. by Amit Gefen. Studies in Mechanobiology, Tissue Engineering and Biomaterials. Berlin, Heidelberg: Springer, 2013, pp. 349–380. ISBN: 978-3-642-36482-2.

[92] Juliano Ferrari Gianlupi. *Script for deploying CompuCell3D simulations as tools in nanoHUB*. Mar. 25, 2022.

[93] Kiri Choi et al. “Tellurium: An extensible python-based modeling environment for systems and synthetic biology”. In: *Biosystems* 171 (Sept. 2018), pp. 74–79.

[94] Juliano Ferrari Gianlupi. *Getting your tellurium project on nanohub*. Nov. 19, 2021.

[95] Juliano Ferrari Gianlupi. *CompuCell3D - 2D wet foam coarsening*. Version 1.3. 2023.

[96] Juliano;Sego Ferrari Gianlupi. “CompuCell3D - Simulation of cell crawling in 3D”. In: (2022).

[97] Juliano;T Ferrari Gianlupi. “CompuCell3D - Bacterium Macrophage”. In: (2019).

[98] Juliano Ferrari Gianlupi. “CompuCell3D - Avascular Tumor Growth and Mutation”. In: (2023).

[99] Juliano;T Ferrari Gianlupi. “CompuCell3D Vascular Tumor”. In: (2019).

[100] Juliano;Sego Ferrari Gianlupi. “CompuCell3D v4 Main Tool”. In: (2020).

[101] Juliano;Sego Ferrari Gianlupi. “COVID-19 drug treatments explorer, CompuCell3D”. In: (2021).

[102] Juliano;Sego Ferrari Gianlupi. “CompuCell3D - Chemotactic Elongation Demo”. In: (2021).

[103] Juliano;Sego Ferrari Gianlupi. “FocalPointPlasticity Plugin Demo”. In: (2021).

[104] Juliano;Glazier Ferrari Gianlupi. “Cancer Evolution in CompuCell3D”. In: (2021).

[105] T. J. ;Aponte-Serrano Sego. “COVID 19 Virtual Tissue Model - Tissue Infection and Immune Response Dynamics”. In: (2020).

[106] Juliano;Sego Ferrari Gianlupi. “CompuCell3d cell sorting in a hexagonal lattice”. In: (2020).

[107] Juliano;Sego Ferrari Gianlupi. “CompuCell3D - Cells random walking at different speeds”. In: (2020).

[108] Juliano;Sego Ferrari Gianlupi. “CompuCell3D - Delta-Notch signaling in a group of cells”. In: (2020).

[109] Juliano;Sego Ferrari Gianlupi. “CompuCell3D - Simulation of angiogenesis”. In: (2020).

[110] Juliano;Sego Ferrari Gianlupi. “CompuCell3D v4 - Bacterium Macrophage simulation”. In: (2020).

[111] Juliano Ferrari Gianlupi. *CompuCell3D - 2D wet foam coarsening with drainage*. Version 1.3. 2023.

[112] *COPASI—a COmplex PAthway SImulator | Bioinformatics | Oxford Academic*.

VITA

Juliano Ferrari Gianlupi

Education

– Ph.D. in Intelligent Systems Engineering

- Indiana University Bloomington, USA
- Concentration: Bioengineering
- Advisor: James A. Glazier
- Duration: Jan 2018 - Aug 2023
- Research Areas: agent-based multiscale modeling of cells and tissues, biological dynamical systems, building infrastructure for nanoBIO.

– M.Sc. in Physics

- Federal University of Rio Grande do Sul, Brazil
- Advisor: Gilberto L. Thomas
- Duration: Jan 2016 - Dec 2017

– B.Sc. in Physics

- Federal University of Rio Grande do Sul, Brazil
- Duration: Jan 2011 - Dec 2015

Research Experience

– Intelligent Systems Engineering

- Projects: Developing agent-based models of cells to predict tissue-level biological function and disease. Focused on modeling the spread of viral infection in tissues

and the cellular immune response. Coupling PK/PKPD simulations with agent-based modeling. Modeling infrastructure development.
- Duration: Jan 2018 - Aug 2023

– Eli Lilly & Co., PK/PD and Pharmacometrics Team

- Project: investigation into a novel way of using available COVID-19 data under the supervision of Dr. Emmanuel Chigutsa, in the PK/PD modeling team.
- Duration: May 2022 - Aug 2022

– Federal University of Rio Grande do Sul, Physics

- Project: soft-materials models to investigate the evolution of dry and wet foams, and how the evolution changes with the liquid fraction
- Duration: Aug 2014 - Dec 2017

Teaching Experience

– Co-Instructor

- Indiana University Bloomington, Luddy SICE-ISE
- Jan 2020 - May 2020; Jan 2021 - May 2023
- Courses: Computational Modeling Methods for Virtual Tissues, Introduction to Computational Bioengineering—Dynamics on Networks

– Workshop Instructor

- 2021 Virtual CompuCell3D User Training Workshop
- 2020 Virtual CompuCell3D User Training Workshop
- 2019 Virtual CompuCell3D User Training Workshop
- 2018 Virtual CompuCell3D User Training Workshop
- 2017 Virtual CompuCell3D User Training Workshop

Publications

– Journal Articles

- Gianlupi, J. F., Sego, T. J., Sluka, J. P., & Glazier, J. A. (2023). PhenoCellPy: A Python package for biological cell behavior modeling. bioRxiv, 2023-04.
- Ferrari Gianlupi, J., Mapder, T., Sego, T. J., Sluka, J. P., Quinney, S. K., Craig, M., ... & Glazier, J. A. (2022). Multiscale model of antiviral timing, potency, and heterogeneity effects on an epithelial tissue patch infected by SARS-CoV-2. Viruses, 14(3), 605.
- Sego, T. J., Aponte-Serrano, J. O., Gianlupi, J. F., & Glazier, J. A. (2021). Generation of multicellular spatiotemporal models of population dynamics from ordinary differential equations, with applications in viral infection. BMC biology, 19(1), 1-24.
- Zarnitsyna, V. I., Gianlupi, J. F., Hagar, A., Sego, T. J., & Glazier, J. A. (2021). Advancing therapies for viral infections using mechanistic computational models of the dynamic interplay between the virus and host immune response. Current Opinion in Virology, 50, 103-109.
- Sego, T. J., Aponte-Serrano, J. O., Ferrari Gianlupi, J., Heaps, S. R., Breithaupt, K., Brusch, L., ... & Glazier, J. A. (2020). A modular framework for multiscale, multicellular, spatiotemporal modeling of acute primary viral infection and immune response in epithelial tissues and its application to drug therapy timing and effectiveness. PLoS computational biology, 16(12), e1008451.
- de Lima, C. F., Gianlupi, J. F., Metzcar, J., & Zerick, J. (2020). Accelerated solving of coupled, non-linear ODEs through LSTM-AI. arXiv preprint arXiv:2009.08278.
- Getz, Michael, et al. "Iterative community-driven development of a SARS-CoV-2 tissue simulator." BioRxiv (2020): 2020-04.

– Conferences

- APS 2023 March Meeting, March 2023, contributed talk

- 12th European Conference on Mathematical and Theoretical Biology, September 2022, contributed talk
- German Conference on Bioinformatics, September 2022, contributed talk
- IMAG/MSM Working Group (Multiscale Modeling and Viral Pandemics), February 2022, invited talk
- 2020 Society of Mathematical Biology, August 2020, Contributed Poster

– Computational Tools

- **Script for deploying CompuCell3D simulations as tools in nanoHUB**: Script to make deployment of CompuCell3D simulations on nanoHUB, `https://github.com/JulianoGianlupi/cc3d-nanoHub-tool-maker`
- **CompuCell3D v4 Main Tool**: Base tool for CompuCell3D version 4 and greater. Includes running any of the demos included with CC3D, `https://nanohub.org/tools/cc3dbase4x`
- **CompuCell3D - Bacterium Macrophage**: online deployment of CompuCell3D simulation of Macrophage hunting bacterium through a maze, `https://nanohub.org/tools/cc3dbactmacro`
- **CompuCell3D - Simulation of cell crawling in 3D**: Online deployment of CompuCell3D simulation of cell crawling in 3D by Fortuna et al. 2020 `https://doi.org/10.1016/j.bpj.2020.04.024`, `https://nanohub.org/tools/gltcellcrawl`
- **CompuCell3D - 2D wet foam coarsening**: CompuCell3D Demo for a 2D foam coarsening without drainage, `https://nanohub.org/tools/cc3dwf`
- **CompuCell3D - 2D wet foam coarsening with drainage**: CompuCell3D Demo for a 2D foam coarsening with drainage, `https://nanohub.org/tools/cc3dwfdrain`
- **CompuCell3D v4 - Bacterium Macrophage simulation**: Macrophage hunting bacterium through a maze using CompuCell3D v4, `https://nanohub.org/tools/bacmacrocc3d4`

- **CompuCell3D - Simulation of angiogenesis**: CompuCell3D simulates angiogenesis using chemical signals, https://nanohub.org/tools/angiogencc3d
- **CompuCell3D - Delta-Notch signaling in a group of cells**: CompuCell3D can solve individual cell's ODE and have the information of one cell affect another (implemented trough SBML), https://nanohub.org/tools/deltanotchcc3d
- **CompuCell3D - Cells random walking at different speeds**: Cells random walking at different speeds implemented through the motility plugin, https://nanohub.org/tools/mot2ddemocc3d
- **CompuCell3d cell sorting in a hexagonal lattice**: Showcases hexagonal lattice use in CompuCell3D by simulating cell sorting by difference in contact energies in a hexagonal lattice, https://nanohub.org/tools/cellsorthexcc3d
- **COVID 19 Virtual Tissue Model - Tissue Infection and Immune Response Dynamics**: Simulates tissue and immune system interactions during a viral lung infection, https://nanohub.org/tools/cc3dcovid19
- **Cancer Evolution in CompuCell3D**: Demonstrates implemetation of cancer evolution in CompuCell3D, https://nanohub.org/tools/cancerevocc3d
- **FocalPointPlasticity Plugin Demo**: This simple demo shows basic functionality of FocalPointPlasticity and how link rigidity affects cell behaviors, https://nanohub.org/tools/cc3dfppdemo
- **CompuCell3D - Chemotactic Elongation Demo**: Implementation of Developmental biology 289.1 (2006): 44-54. and PLoS Comput Biol 4.9 (2008): e1000163, https://nanohub.org/tools/cc3delongdemo
- **COVID-19 drug treatments explorer, CompuCell3D**: Explores possible drug treatments for COVID-19; namelly viral entry inhibition and viral replication inhibition, https://nanohub.org/tools/coviddrugexp
- **CompuCell3D Vascular Tumor**: Simulate 3D vascular tumor with CompuCell3D, https://nanohub.org/tools/cc3dvasctumor
- **CompuCell3D - Avascular Tumor Growth and Mutation**: vascular tumor, growing on nutrient, self-limited, mutates, https://nanohub.org/tools/avas

ctum

Campus Activities

– Indiana Graduate Workers Coalition: Union Representative

- Jan 2022 - Dec 2022

– Indiana Graduate Workers Coalition: Social Media Manager

- Jan 2022 - Dec 2022

Languages

- Portuguese, native
- English, fluent
- Spanish, advanced

Skills

- Python
- R
- NONMEM
- COPASI
- Fortran 90
- C++
- PyThorch
- TensorFlow
- CompuCell3D
- PhysiCell
- Git
- LaTeX