

COMPUTATIONAL APPROACHES TO *D. RERIO* RETINAL ORGANOGENESIS

by

Michael Mattocks

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Cell and Systems Biology
University of Toronto

Abstract

Computational Approaches to *D. rerio* Retinal Organogenesis

Michael Mattocks

Doctor of Philosophy

Graduate Department of Cell and Systems Biology

University of Toronto

2020

Increasing integration of sophisticated statistical and computational techniques into the analysis of cell and molecular biological data has created many opportunities for explaining organogenic phenomena by numerical analysis. This thesis first examines the suitability of the most developed of these explanations for the development of the *D. rerio* eye, and demonstrates that it is neither formally correct nor explanatory. By documenting the phenomena associated with postembryonic retinal neurogenesis, a framework of observations which cellular models of this process would be called upon to explain is generated. The desiderata of a model comparison framework suitable for evaluating the evidence supplied by confocal datasets from postembryonic fish are explicated. Nuclear dynamics in a mutant fish, *rys*, are

This work is dedicated to the memory of my brother Gareth Akerman.

Acknowledgements

Acknowledgements text

Contents

Introductory Notes	1
I Modelling Studies	2
Précis of Modelling Studies	3
1 Canonical retinal progenitor cell phenomena and their explanations	5
1.1 The Harris Stochastic Mitotic Mode Explanation (SMME)	5
1.2 Explanations for RPC function in 2009 and the drive to unification	6
1.3 Canonical vertebrate RPC phenomena: the RPC “morphogenetic alphabet”	7
1.3.1 Proliferative phenomena	8
1.3.2 Specificative phenomena	9
1.3.3 Other morphogenetic phenomena	11
1.4 Macromolecular mechanistic explanations for RPC phenomena	12
1.4.1 Transcription factor networks	12
1.4.2 Intercellular signalling networks	13
1.4.3 Patterning mechanisms	14
1.4.4 Chromatin dynamics	15
1.5 A unified theory of RPC function? “Blurring” to order	15
1.6 Explanatory Strategy and Intent of the SMME	17
2 “Stochastic mitotic mode” models do not explain zebrafish retinal progenitor lineage outcomes	19
2.1 Introduction	19
2.2 Results and Discussion	21
2.2.1 Gomes SSM: Ancestral Model of the SMME SSMs	22
2.2.2 He SSM: Explaining variability in zebrafish neural retina lineage size	22
2.2.3 Boije SSM: Explaining variability in zebrafish RPC fate outcomes	25
2.2.4 Model selection demonstrates the SMME is not the best available explanation for RPC lineage outcomes	26
2.2.5 SMME SSMs cannot explain the post-embryonic phase of CMZ-driven zebrafish retinal formation	29
2.3 Conclusion	33

3 Toward a computational CMZ model comparison framework	36
3.1 SMME Postmortem: a wrong turn at Gomes	36
3.2 Implications of the SMME's failure for modelling CMZ RPCs	38
3.3 Desiridata for spatial CMZ models in a putative model comparison framework	40
3.3.1 Spatial dimension of the models: the "slice model"	40
3.3.2 Temporal resolution of the models	41
3.4 Bayesian decisionmaking for structuring models under uncertainty	42
4 Bayesian periodization of postembryonic CMZ activity	44
4.1 Preliminaries: Calculation of Bayesian evidence militates for independent log-Normal modelling of CMZ parameters	44
4.2 Survey of CMZ population and gross retinal contribution	48
4.3 Periodization of postembryonic CMZ activity by Galilean Monte Carlo Nested Sampling .	50
4.4 Slice-model characterisation of asymmetrical CMZ population dynamics by Galilean Monte Carlo Nested Sampling	53
4.4.1 Cumulative thymidine labelling supports GMC-NS model predictions	59
4.4.2 Asymmetry in dorso-ventral cohort contributions highlights exit rate parameter slope .	59
4.5 Investigating CMZ RPC lineage outcomes	59
4.6 Microglial apoptotic fate of <i>D. rerio</i> retinal neurons & functional significance of CMZ activity	63
5 Mutant npat results in nucleosome positioning defects in <i>D. rerio</i> CMZ progenitors, blocking specification but not proliferation	64
5.1 Introduction	64
5.2 Results	66
5.2.1 The <i>rys</i> CMZ phenotype is characterised failure of RPCs to specify, altered nuclear morphology, aberrant proliferation and expanded early progenitor identity	66
5.2.2 The microphthalmic zebrafish line <i>rys</i> is an npat mutant	72
5.2.3 <i>rys</i> siblings and mutants have unique sets of nucleosome positions, best explained by different sequence preferences and increased sequence-dependent positioning in mutants	77
5.3 Discussion	84
6 Inferring and modelling nuclear dynamics in retinal progenitors	88
6.1 Intro	88
6.2 Frontiers of nuclear dynamics	88
6.3 Integrating nuclear dynamics into models of CMZ- abstraction and biosemiotics	88
II Software Technical Reports	89
7 GMC_NS.jl	90
7.1 Implementation notes	90
7.2 Ensemble, Model, and Model Record interfaces	91
7.3 Usage notes	92

7.3.1	Setting up for a run	92
7.3.2	Parallelization	92
7.3.3	Displays	92
7.3.4	Example use	92
8	BioBackgroundModels.jl	93
8.0.1	Genome partitioning and sampling	93
8.0.2	Optimizing BHMMs by EM algorithms	93
8.0.3	Displaying results	93
9	BioMotifInference.jl: Independent component analysis motif inference by nested sampling for Julia	94
9.1	Introduction	94
9.2	Implementation of the nested sampling algorithm	94
9.3	Usage notes	96
9.4	Recovery of spiked motifs	96
III	Supplementary Materials	97
10	Supplementary materials for Chapter 2	98
10.1	Materials and methods	98
10.1.1	Zebrafish husbandry	98
10.1.2	Proliferative RPC Histochemistry	98
10.1.2.1	Anti-PCNA histochemistry	98
10.1.2.2	Cumulative thymidine analogue labelling for estimation of CMZ RPC cell cycle length	99
10.1.2.3	Whole retina thymidine analogue labelling of post-embryonic CMZ contributions	99
10.1.3	Confocal microscopy and image analysis	99
10.1.4	Estimation of CMZ annular population size	100
10.1.5	Modelling lens growth	100
10.1.6	Estimation of 3dpf CMZ cell cycle length	100
10.1.7	CHASTE Simulations	101
10.1.7.1	Project code	101
10.1.7.2	SPSA optimisation of models	101
10.2	Supporting figures	103
11	Supplementary materials for Chapter 4	108
11.1	Description of the computational cluster used in the work	108
11.2	Developmental progression of naso-temporal population asymmetry in the CMZ	108
11.3	Cumulative EdU Bayesian Linear Regression	109
11.4	Evidence calculations for Normal and Log-Normal models of layer and lineage contribution	111
11.5	Likelihood ratio calculations for Normal and Log-Normal models of layer and lineage contribution	111

11.6 Methods	111
11.6.1 Statistical Analyses	111
12 Supplementary material for Chapter 5	112
12.1 Synteny analysis of genomic surroundings of <i>D. rerio</i> npat	112
12.2 10dpf <i>rys</i> RPCs are mitotic	112
12.3 The fate of <i>rys</i> RPCs is microglial phagocytosis	112
12.4 PWM sources detected in the combined sib and <i>rys</i> differential position set	112
13 Theoretical Appendix A: Model theory and statistical methods	117
13.1 Model Theory	117
13.1.1 Bayesian Epistemological View on Model Comparison	118
13.1.2 Model sampling and optimization	120
13.1.3 Overfitting	120
13.1.4 Monte Carlo simulation	120
13.1.5 Simple Stochastic Models	120
13.2 Statistical Methods	123
13.2.1 Bayesian parameter estimation	123
13.2.1.1 Problems with frequentist inference using normal models of sample data .	123
13.2.1.2 The Bayesian approach to normal models of unknown mean and variance	123
14 Theoretical Appendix B: Metaphysical Arguments	125
14.1 Chance is not a valid explanation for biological phenomena; Randomness is a measure of property of sequences and not an explanation	125
14.1.0.1 Chance versus Randomness	126
14.1.0.2 Chance in molecular mechanisms	127
14.1.0.3 Can macromolecular chanciness be rooted in quantum indeterminacy? .	129
14.1.0.4 Randomness in RPC fate specification	130
14.1.0.5 Summary: “Stochastic” or “variable”?	130
14.2 Macromolecular mechanistic explanations in the Systems era	131
14.2.1 The Feyerabendian modeller	133
15 Code Appendix	136
15.1 SPMM	136
15.1.1 /apps/src/BoijeSimulator.cpp	136
15.1.2 /apps/src/GomesSimulator.cpp	142
15.1.3 /apps/src/HeSimulator.cpp	149
15.1.4 /apps/src/WanSimDebug.cpp	160
15.1.5 /apps/src/WanSimulator.cpp	164
15.1.6 /python_fixtures/He_output_fixture.py	171
15.1.7 /python_fixtures/Kolmogorov_fixture.py	179
15.1.8 /python_fixtures/SPSA_fixture.py	185
15.1.9 /python_fixtures/Wan_output_fixture.py	202
15.1.10 /python_fixtures/diagram_utility_scripts/Gomes_He_cycle_plots.py .	206
15.1.11 /python_fixtures/diagram_utility_scripts/He_Boije_signal_plots.py .	207

15.1.12 /python_fixtures/figure_plots/Cumulative_EdU.py	209
15.1.13 /python_fixtures/figure_plots/He_output_plot.py	211
15.1.14 /python_fixtures/figure_plots/Kolmogorov_plot.py	230
15.1.15 /python_fixtures/figure_plots/Mitotic_rate_plot.py	231
15.1.16 /python_fixtures/figure_plots/Wan_output_plot.py	235
15.1.17 /src/BoijeCellCycleModel.cpp	236
15.1.18 /src/BoijeCellCycleModel.hpp	245
15.1.19 /src/BoijeRetinalNeuralFates.cpp	251
15.1.20 /src/BoijeRetinalNeuralFates.hpp	252
15.1.21 /src/GomesCellCycleModel.cpp	256
15.1.22 /src/GomesCellCycleModel.hpp	265
15.1.23 /src/GomesRetinalNeuralFates.cpp	270
15.1.24 /src/GomesRetinalNeuralFates.hpp	272
15.1.25 /src/HeAth5Mo.cpp	277
15.1.26 /src/HeAth5Mo.hpp	277
15.1.27 /src/HeCellCycleModel.cpp	279
15.1.28 /src/HeCellCycleModel.hpp	293
15.1.29 /src/OffLatticeSimulationPropertyStop.cpp	299
15.1.30 /src/OffLatticeSimulationPropertyStop.hpp	307
15.1.31 /src/WanStemCellCycleModel.cpp	313
15.1.32 /src/WanStemCellCycleModel.hpp	321
15.2 BioBackgroundModels	326
15.2.1 /README.md	326
15.2.2 /src/BioBackgroundModels.jl	326
15.2.3 /src/API/EM_master.jl	327
15.2.4 /src/API/genome_sampling.jl	331
15.2.5 /src/API/reports.jl	334
15.2.6 /src/BHMM/BHMM.jl	335
15.2.7 /src/EM/EM_converge.jl	337
15.2.8 /src/EM/baum-welch.jl	338
15.2.9 /src/EM/chain.jl	342
15.2.10 /src/EM/churbanov.jl	343
15.2.11 /src/genome_sampling/partition_masker.jl	346
15.2.12 /src/genome_sampling/sequence_sampler.jl	354
15.2.13 /src/likelihood_funcs/bg_lh_matrix.jl	364
15.2.14 /src/likelihood_funcs/hmm.jl	367
15.2.15 /src/reports/chain_report.jl	368
15.2.16 /src/reports/partition_report.jl	373
15.2.17 /src/reports/replicate_convergence.jl	375
15.2.18 /src/utilities/BBG_analysis.jl	379
15.2.19 /src/utilities/BBG_progressmeter.jl	380
15.2.20 /src/utilities/HMM_init.jl	383
15.2.21 /src/utilities/load_balancer.jl	384

15.2.22 /src/utilities/log_prob_sum.jl	385
15.2.23 /src/utilities/model_display.jl	385
15.2.24 /src/utilities/observation_coding.jl	386
15.2.25 /src/utilities/utilities.jl	389
15.2.26 /test/ref_fns.jl	390
15.2.27 /test/runtests.jl	394
15.2.28 /test/synthetic_sequence_gen.jl	419
15.3 BioMotifInference	422
15.3.1 /README.md	422
15.3.2 /src/BioMotifInference.jl	422
15.3.3 /src/IPM/ICA_PWM_Model.jl	424
15.3.4 /src/IPM/IPM_likelihood.jl	429
15.3.5 /src/IPM/IPM_prior_utilities.jl	436
15.3.6 /src/ensemble/IPM_E ensemble.jl	438
15.3.7 /src/ensemble/ensemble_utilities.jl	444
15.3.8 /src/nested_sampler/converge_ensemble.jl	448
15.3.9 /src/nested_sampler/nested_step.jl	452
15.3.10 /src/permuation/Permute_Tuner.jl	455
15.3.11 /src/permuation/orthogonality_helper.jl	458
15.3.12 /src/permuation/permute_control.jl	461
15.3.13 /src/permuation/permute_functions.jl	466
15.3.14 /src/permuation/permute_utilities.jl	482
15.3.15 /src/utilities/model_display.jl	488
15.3.16 /src/utilities/ns_progressmeter.jl	492
15.3.17 /src/utilities/worker_diagnostics.jl	498
15.3.18 /test/runtests.jl	499
15.3.19 /test/spike_recovery.jl	500

List of Tables

2.1	AIC values for models assessed against training and test datasets	29
4.1	Likelihood ratio comparison between normal and log-normal models of retinal population parameters	45
4.2	Evidence favours log-normal models of retinal population parameters	45
4.3	Evidence favours uncorrelated linear models of CMZ-population and retinal volume over time	47
4.4	Evidence favours a 2-phase periodization of CMZ activity	52
4.5	Maximum a posteriori parameter estimates for periodization models	52
4.6	Evidence favours a combined slice model over separate dorsal and ventral models	56
4.7	Maximum a posteriori parameter estimates for slice models	57
4.8	Evidence favours whole-CMZ linear cycle models over separate D/V models .	59
4.9	Evidence supports stable model of retinal contributions	63

List of Figures

1.1	Histogenetic birth order of retinal neurons	10
2.1	Structure of Gomes SSM	23
2.2	Structure of He SSM	24
2.3	Structure of Boije SSM	25
2.4	Model comparison: the SPSA-optimised He SSM and a deterministic alternative	28
2.5	Most zebrafish retinal neurons are contributed by the CMZ between one and three months of age	30
2.6	Per-lineage probabilities of mitoses, He et al. observations compared to model output	31
2.7	CMZ population of proliferating RPCs: estimates from observations and simulated Wan-type CMZs	33
4.1	CMZ population and retinal volume estimates are uncorrelated at 3dpf	47
4.2	Population and activity of the CMZ over the first year of <i>D.rerio</i> life	49
4.3	Maximum a posteriori output of periodization models	51
4.4	Kernel density estimates of marginal posterior parameter distributions, 2-phase model	54
4.5	Developmental progression of dorso-ventral population asymmetry in the CMZ	55
4.6	Maximum a posteriori output of total, dorsal, and ventral CMZ slice models	57
4.7	Kernel density estimates of marginal posterior parameter distributions, total slice model	58
4.8	Representative 23dpf lineage marker confocal micrographs	61
4.9	CMZ contributions to the neural retina over time by layer and lineage marker	62
5.1	<i>rjs</i> mutants exhibit a small-eye phenotype	65
5.2	<i>rjs</i> CMZ populations start relatively small and quiescent, end abberantly large and proliferative	66
5.3	<i>rjs</i> CMZ RPCs fail to contribute to the neural retina	67
5.4	7dpf <i>rjs</i> CMZ RPCs display enhanced asymmetry, increased volume, decreased sphericity, and relative but not absolute enlargement	69
5.5	RPC nuclei of the <i>rjs</i> CMZ display disorganized, loosely packed chromatin	70
5.6	<i>rjs</i> mutant CMZs have increased caspase-3 positive nuclei	71
5.7	Mutant <i>rjs</i> RPCs display expanded expression of early progenitor markers	72

5.8 RT-PCR analysis reveals two aberrant intron retention variants in <i>rys</i> mutant npat transcripts	73
5.9 Functional domains of Human NPAT compared to predicted wild-type and <i>rys</i> <i>Danio</i> npat	74
5.10 In situ hybridization reveals progressive restriction of npat expression to the CMZ	75
5.11 <i>rys</i> overexpress total and polyadenylated core histone transcripts	76
5.12 48hpf embryos injected with an npat ATG-targeted morpholino display a small-eye phenotype	77
5.13 <i>rys</i> chromosomes are differentially enriched and depleted of nucleosome position density and occupancy.	79
5.14 Novel nucleosome positions in <i>rys</i> occur in similar numbers to those lost from sibs.	80
5.15 PWM sources detected in sibling differential nucleosome positions.	82
5.16 PWM sources detected in <i>rys</i> mutant differential sources.	83
 11.1 Developmental progression of naso-temporal population asymmetry in the CMZ.	109
11.2 Linear regressions performed on cumulative labelling data from dorsal, ventral, and combined CMZ sectional populations	110
 12.1 Synteny Database output for the syntenic region containing <i>D. rerio</i> npat	113
12.2 Rys CMZ RPCs are mitotic at 10dpf	114
12.3 4C4-positive microglia engulf <i>rys</i> mutant RPCs	115
12.4 PWM sources detected in combined differential sources.	116
 13.1 Simple stochastic stem cell model, representing probabilities of cell division events, excerpted from Fagan 2013 pg. 61. Black circles denote proliferative cells, while white and grey circles denote different types of postmitotic offspring. “Number of progeny in P” is the number of mitotic offspring produced by each type of division. The probability of each division type must sum to 1, as all possibilities are represented, granting that the division types are defined by the postdivisional mitotic history of the offspring.	121
 14.1 Cellular systems model-construction, excerpted from [Fag15, p.7]. The system of equations and subsequent steps are based on a 2-element wiring diagram.	132

Introductory Notes

Thesis Guide

This document is split into three parts. Part I contains results and discussion pertaining to empirical modelling studies that will be of interest to committee members and developmental biologists. Part II contains technical reports pertaining to novel software that was written in order to perform the analyses presented in Part I. Part III contains a variety of supplementary materials arising from Parts I and II. These include detailed descriptions of the methods used in Part I, less-technical explanations of relevant statistical and model theory, the source code of all software and analyses, and the bibliography. The source code has been omitted from the print document.

Readers of the .pdf document will find some text unobtrusively highlighted in plum throughout the thesis. Section indices highlighted in this way link to the specified section. Technical terms have also been highlighted when pertinent material is available in Part III to explain them for the reader who may be unfamiliar.

Terminology and Style

Throughout [Chapter 1](#), [Chapter 2](#), and [Chapter 3](#), I have used the appellation "Harris" when referring to the output of William Harris' research group. This is a large body of research spanning several decades, and involves many co-authors. My use of "Harris" here is a convenience, as Harris is the only common author across the period in question, and presumably the agent carrying these ideas forward from project to project. It is not intended to slight or minimize the contributions of any of the other members of Harris' group (many of whom are now senior scientists in their own right). I have used "Raff" in an identical sense in referring to the Raff group's work in [Chapter 3](#).

I have preferred the terms "specification", "determination", and their derivatives, to refer to cells assuming a particular lineage fate. "Differentiation" is well-understood, but ambiguous, and often understood to relate to the mitotic event itself, which I generally do not intend. "A cell has specified" means it has assumed a stable macromolecular identity or "fate". This term is intended to correspond exactly with the appearance of stable markers of cell type.

There are a number of specialised philosophical terms that appear in quoted material in the Part III. I have made occasional use of some of them to save space. These are as follows:

iff - "if and only if"

explanadum - the fact to be explained

explanans - the explanation itself

Unless otherwise noted, formatting of quoted material is preserved, so that italic emphases appear in the original. An exception to this is citations in original material, which have been removed wherever irrelevant or uninformative (ie. numbered references). I have indicated my own editorial comments and alterations to quoted material with [square braces].

Part I

Modelling Studies

Précis of Modelling Studies

The primary concern of this thesis is developing and assessing computational models of retinal progenitor cell (RPC) phenomena. While many different, well substantiated explanations for various aspects of eye development exist, these explanations have usually been supported by assessing the frequentist significance of observed effects, not by building formally testable models of the phenomena themselves. Of the explanations which could be subject to model comparison, virtually all of these are derivatives of the Simple Stochastic Model (SSM), dating to the earliest work, performed by Till and McCulloch [TMS64].

The explosion of molecular biological information has massively increased the number of parameters that might be included in RPC models. The question of how RPC activity might be successfully predicted and controlled *in vivo* may revolve around finding adequate models that explain retinogenetic processes. The development and assessment of such models is, therefore, of fundamental basic and applied interest. This is particularly so, in light of increasing interest in entraining RPCs for the purpose of retinal regenerative medicine. Our group is particularly interested in the possibility that RPCs located in the peripheral retinal annulus of the circumferential marginal zone (CMZ), which are present in zebrafish and mammals alike, could be harnessed for retinal repair.

It is surprising that, despite the use of SSMs, the literature on neural stem cells contains virtually no systematic statistical approaches to the construction, optimization, or comparison of models. This thesis is an attempt to address this problem. It proceeds in three basic strokes. Firstly, Chapters 1 and 2 evaluate the existing explanations of RPC behaviour, and find that those with formal model components are deficient, and cannot explain the activities of RPCs in the CMZ. Secondly, Chapters 3 and 4 propose a general CMZ modelling framework under which the Bayesian evidence for different model-hypotheses might be assessed, and, by testing a variety of population-level hypotheses, provide guidance and develop the methods required to test cell-based hypotheses. Lastly, Chapters 6, 5, and ?? introduce the zebrafish CMZ mutant *rys*, and apply some of the methods developed in the second stroke to explain aspects of the aberrant morphology and behaviour of *rys* RPCs.

Chapter 1 begins by summarizing the range of explanations that have been offered for RPC activities, and introduces the primary set of formal models that have recently been used to favour an explanation centered around a supposed stochastic mitotic mode of RPCs. Chapter 2 elucidates the structure and development of the stochastic mitotic mode explanation (SMME) models, finding fundamental logical errors in their interpretation of "stochasticity". Moreover, by pursuing a basic information theoretical model selection approach, it is demonstrated that the SMME is not even the best available explanation relying on this flawed notion of stochasticity.

Chapter 3 discusses the implications of Chapter 2 for models of RPCs generally, and in the CMZ particularly; it also recommends a better model selection approach from Bayesian theory, nested sampling. Chapter 4 surveys the activity of RPCs in the post-embryonic zebrafish CMZ, developing Bayesian methods for doing so. These methods include the use of nested sampling to solve the long-standing problem of estimating cell cycle parameters of subpopulations of cells from cumulative thymidine labelling measurements of the entire super-population. It concludes with recommendations for future modelling approaches to the CMZ, structured by the foregoing Bayesian analysis.

Chapter 5 identifies the causative mutation in *rys* and shows that the zebrafish CMZ mutant *rys* CMZ RPCs have disorganized chromatin, likely blocking differentiation but not proliferation. Nested sampling is used to demonstrate that the nuclear phenotype is likely the consequence of a shift in

nucleosome histone composition in *rys* progenitors. ?? discusses some of the theoretical issues raised by the *rys* analysis, and their implications for modelling retinogenesis more broadly, especially in integrating nuclear dynamics into models of RPCs.

Chapter 1

Canonical retinal progenitor cell phenomena and their explanations

1.1 The Harris Stochastic Mitotic Mode Explanation (SMME)

The work presented in [Chapter 2](#) comprises a critical examination of the best-developed theory of *D. rerio* retinal progenitor cell (RPC) function, and a broader, global general modelling approach motivated by the examination’s findings. The theory in question originates with preëminent retinal biologist William Harris’ research group, referred to hereafter Stochastic Mitotic Mode Explanation (SMME). This theory purports to explain the function of zebrafish RPCs in terms of the stochastic effects of two particular transcription factors.

The SMME for RPC function is of fundamental theoretical and practical interest. It arises from a concerted effort by the Harris group to make sense of the difficult problem of explaining how a complex tissue like a retina can arise from a field of similar proliferating cells. In 2009, Harris coauthored a detailed review chapter, documenting the bewildering array of macromolecules and cellular processes thought to be involved in RPC function [\[AH09\]](#). This enumeration contains many caveats and notes that the effects of particular macromolecules routinely differ between developmental stages, cell types, organisms, and so forth. At this time, no clear, detailed, comprehensive models of RPC function had been advanced, and the review is typified by statements like “It is difficult to reconcile all the studies on the initiation and spread of neurogenesis in a single model.”¹ It is therefore remarkable that, over the next nine years, the Harris group would go on to promulgate a [simple stochastic model](#) of zebrafish RPC function invoking only two named macromolecules.

Harris thus seems to be making a bold attempt to cut the Gordian knot of conflicting evidential threads and advance a simple, comprehensible, “mind-sized” model of RPC function. In effect, Harris’ explanation for zebrafish RPC function is a microcosm of the broader promise of “Systems Biology” to make sense of the contemporary welter of conflicting datasets. By using sophisticated mathematical methods drawn from information and complexity theories, the apparent confusion will be clarified and underlying molecular mechanisms will be revealed. Given Harris’ track record and preeminence in the

¹This was in no sense a problem with Harris’ understanding. A similarly high-level review coauthored by Pam Raymond [\[AR08\]](#) concluded, with regard to models of photoreceptor fate specification: “The data reviewed in the preceding sections indicate that a ‘one-size-fits-all’ model is not possible...”

field, we have good reason to take seriously the possibility that he has succeeded. Examining whether this is the case is our first priority. In order to appreciate the scope of his theoretical maneuver, and possible alternatives to it, it is necessary to begin by summarising the state of the art at the time of his 2009 review (as well as relevant subsequent additions).

1.2 Explanations for RPC function in 2009 and the drive to unification

In many ways, molecular biologists have been attempting to explain the same remarkable features of retinal progenitor cells for decades. Animal retinas, having relatively well-understood functions and highly stereotypical structures, seem like highly tractable tissues for typical molecular biological explanations. With well defined cell types present in tightly regulated, neurotopologically limited proportions and organisations, it is no surprise that both theoretically-inclined molecular biologists and clinically-inclined regenerative medical practitioners have been keenly interested in the retina². The high level of regular, easily detected order in eyes seemed to suggest a similar level of order and regularity in the macromolecular processes which underlay the formation of the tissue. Retinal biologists have long offered explanations suggesting that RPCs are more-or-less identical and go through highly stereotypical macromolecular processes. It is, however, the persistently observed departures from this (by now, obviously naïve) conception that have occupied most of our attention.

We may crudely gather the RPC-related phenomena studied by biologists under the headings of proliferation, specification, and organisation. A molecular biologist guided by a preference for cognitive simplicity (that is, Occam’s Razor) will reasonably suppose that the simplest explanation for the regularity of retinal development is that any given RPC is executing the same strict “developmental program” as its neighbours. In other words, any one RPC lineages’ proliferative and specificative outcomes are the same as any other RPC lineage. If every progenitor produces a similar number of cells, and the progeny are specified in similar proportions, well-understood principles of cellular adhesion could understandably give rise to the characteristic organisation we observe in animal retinas. However, by the 1980s, vertebrate lineage tracing experiments were routinely revealing a surprising degree of inter-lineage variability in many neural progenitor systems, not least of which was the retina. In their seminal 1987 paper, David Turner and Connie Cepko, using retroviral lineage labelling techniques, demonstrated that individual RPC lineages in rats had a wide range of proliferative and specificative outcomes [TC87]; Harris’ group confirmed this result in *Xenopus* the subsequent year [HBEH88], suggesting this variability was a common feature of vertebrate RPC function.

Indeed, at this point, we find fairly clear accounts of what retinal biologists took their theoretical options to be in explaining this variability. As Harris’ 1988 report states:

Changes in cell character associated with cell type diversification may be controlled in an autonomous way, reflecting either a temporal program inside the cell (Temple and Raff, 1986), the asymmetrical segregation of cytoplasmic determinants (Strome and Wood, 1983; Sulston and Horvitz, 1977), stochastic events inside the cell (Suda et al., 1984), or some combination of these processes. Alternatively, cell type may be controlled in a nonautonomous way, as in cases in which the extracellular environment (Doupe et al., 1985) or cellular interactions (Ready et al., 1976) elicit or limit cell fate. With its multiplicity of cell types, the vertebrate

²Indeed, if central neuroregenerative medicine is to become a clinical possibility, it seems likely that the theoretical and practical issues are most likely to be resolved in eyes before other areas of the CNS.

nervous system would seem to require the ultimate sophistication in its means of cellular determination. [HBEH88]

It is striking, then, that Harris' review of the literature two decades later describes the situation similarly:

Once differentiation is initiated, regulatory mechanisms within the retina ensure that progenitors retain the capacity to undergo more divisions, in parallel with churning out differentiated cells, and that progenitors cease dividing at variable times. There is still debate about the extent of early programming that allows progenitors to step through a series of stereotypical divisions and the extent of regulation from within the whole retina. The production of differentiated cells alters the retinal environment with time...

Moreover, cells from the same clone do not all differentiate at the same time, suggesting three possibilities: a stochastic mechanism for the decision to differentiate, exposure of the two daughters to different environments, or asymmetric inheritance of determinants. [AH09]

In the same paper, he states that the “simple structure and accessibility of the retina make it a useful model to study cell division and differentiation, and as a result most aspects of this have been studied, from lineage tracing of progenitors, to the morphological aspects of division, to the molecular mechanisms involved.” Thus, by Harris' own account, some twenty years of additional research into almost every variety of macromolecular explanation for a huge range of RPC-related phenomena had not provided any means to narrow down the possibilities he had already laid out in 1988. We still have Raff's temporal program (“early programming … step[ping] through a series of stereotypical divisions”), asymmetric segregation of cytoplasmic inheritants during mitotic events, “stochastic” events internal to the cells, and possible “environmental” extracellular determinants. While the number of particular macromolecules functionally implicated in proliferative, specificative, and organisational RPC phenomena had greatly expanded, this had not provided any means to differentiate between these theoretical options. This is only one example of a general phenomenon experienced by molecular biologists, in which enumerationist research programs, directed at producing more and more facts about macromolecular involvement in cellular phenomena, have failed to generate additional theoretical understanding [Kan06]. Harris' SMME therefore represents an example of a “Systems” biological explanation, in which biologists apply the analytical and interpretative methods of the physical and mathematical sciences in an effort to resolve the problems posed by biological complexity [Mor09].

Before proceeding to the SMME itself, let us briefly summarise the diversity of phenomena implicated in RPC function by 2009, as well as the panoply of mechanisms offered as explanations. In doing so, it will become clear what has been elided in the SMME, and what may need to be restored in any alternative modelling approach.

1.3 Canonical vertebrate RPC phenomena: the RPC “molecular genetic alphabet”

The majority of our knowledge of RPC behaviour stems from histological observation employing a limited number of techniques. Simple observations of mitotic figures in a variety of animals had, by the 1950s, revealed the surprising diversity of RPC proliferative phenomena across vertebrate clades. However, it was the advent of lineage tracing techniques, particularly those marking single clonal lineages in whole

retinae, and the extensive use of these techniques in the 1980s-90s, that formed most of the basic body of observations that any macromolecular explanation is now called upon to account for.

Since the majority of vertebrate retinas of biomedical interest are mammalian, and these retinae are fully formed in an early developmental period, RPC behaviour has been best-studied in an embryonic and early developmental context. Here, vertebrate RPCs are derived from the eye field population of the early neural plate and later neural tube. This population is separated into left and right eye primordia, which in turn pouch outwards toward the ectoderm, and, in conjunction with the lens placode (itself induced from the ectoderm), form the optic vesicle. The primitive eye is formed when this vesicle completes a complex morphological folding process, resulting in the cup-shaped structure of the retina [Cav18]. During this process, the cells of the neural retina are differentiated from the overlying retinal pigmented epithelium (RPE). Sometime after the formation of the retinal cup, RPCs begin to exit the cell cycle and are specified as retinal neurons. Studies of this early period revealed numerous difficult-to-explain features of RPC behaviour. Following Larsen's observation that tissue form is attributable to only six behaviours [Lar92], which she describes as the "morphogenetic alphabet", I have categorised RPC phenomena as relating to proliferation, specification, migration, growth, death, and extracellular matrix formation³. Since the vast majority of reported phenomena fall under the first two categories, those belonging to the last four are described collectively.

1.3.1 Proliferative phenomena

Clonal lineage tracing experiments have reliably found that vertebrate RPCs give rise to highly variable numbers of offspring over the collective proliferative "lifetime" of their descendants. The most dramatic of these findings found that rat RPC lineage sizes vary across two orders of magnitude *in vivo*, from 1 to over 200 [TSC90]⁴. The physical organisation of these clones is complex; as detailed below, RPC progeny may appear in any of the 3 retinal layers in a wide variety of specified fate combinations, and may engage in short-range migrations to appropriate positions for their specified fates. Most clonal lineages are "extinguished"; that is, after some time, all of its members have become postmitotic. However, it has long been noted that not all cells produced by RPCs are strictly postmitotic neurons; specified Müller glia retain the ability to reenter the cell cycle in response to stimuli (normally, retinal damage) [DC00, FR03], and peripheral CMZ RPCs remain proliferative in those vertebrates whose eyes grow beyond early development (notably in frogs and fish, while the chick retina has a CMZ of more limited output [FR00]). Therefore, some clonal lineages may be organised into clumps associated with Müller responses, while others in frogs and fish may continue to be "plated out" in a more-or-less linear manner at the retinal periphery for as long as the lineage "lives" [CHW11]. This ongoing RPC contribution to peripheral neurogenesis has long been recognised, so that by 1954 we find the following remarkable statement casually introducing a study of unusual mitoses in the retina of a deepsea fish:

It is conventional⁵ to hold that the growth of the vertebrate retina is only possible due to the presence, in this tissue, of a peripheral germinal zone. In this region, young elements

³I have renamed Larsen's categories to clarify them for the retinal context, but retained her conceptual scheme, which is discussed in more detail in the Theoretical Appendix.

⁴While at least some of this variability must be related to differential integration of lineage markers into "older" (giving fewer offspring) and "newer" (giving more) RPCs, it is generally accepted that vertebrate RPC lineages tend to vary in size by at least one order of magnitude, from less than ten to tens of neurons.

⁵Unfortunately, I am unable to locate the source of this convention, likely due to the poor preservation of many of these older reports. That this required no citation in 1954 suggests the original observations of CMZ proliferation may be in the early 20th century.

actively multiply, and, by subsequent differentiation, give rise to the diverse nervous and sensory constituents of the retina. [VL54]

[author's translation from the French]

Despite this, the proliferating RPCs in this “peripheral germinal zone” (also known as the “ciliary marginal zone”, or CMZ, for its proximity to the retinal ciliary body) have not received the same level of attention as those associated with the central retina. As a consequence, these RPCs have generally, and in particular by Harris, been treated as though they are a type of “frozen” progenitor population, recapitulating spatially, along the peripheral-central axis, the process which RPCs in the central retina undergo in a time-dependent fashion⁶ [HP98].

The length of the RPC cell cycle has been of considerable interest, since the evolution of this parameter in time, in conjunction with the RPC population size (the number of cells specified in the eye field), determines the eventual size of differentiated retinal neural population, and therefore the retina. RPC cell cycle length has generally been inferred from clonal lineage size, although it has also been occasionally assayed directly in cumulative thymidine analogue labelling experiments. Vertebrate RPCs have generally been found to undergo a period of relative quiescence, in which the cell cycle lengthens, before the neural retina begins to be specified (in zebrafish, this period is 16-24 hpf). The cell cycle shortens as RPCs begin to exit the cell cycle [HH91, LHO⁺00]. After the central retina is specified, the RPC cell cycle once again lengthens, and is presumed to continue to slow until RPCs have completed differentiation⁷.

Finally, the orientation of the RPC division plane in mitosis has also been implicated in retinal organisation. The orientation of divisions has been associated both with the proliferative and differentiative fate of RPC progeny. For instance, interfering with spindle orientation in the developing rat retina, such that more RPC divisions occur parallel to the neuroepithelial plane (rather than along the apico-basal axis) results in more proliferative and fewer postmitotic, specified progeny[ŽCC⁺05]. That said, it seems that whatever effects are attributed to mitotic orientation are likely species-specific, as zebrafish RPCs display a different pattern of axis orientation, dividing mainly in the epithelial plane[DPCH03].

1.3.2 Specificative phenomena

Irrespective of their location in the retina, vertebrate RPC lineages have offspring which may enter any of the three cellular layers of the retina. Moreover, single lineages can include any possible combination of cell fates, so that RPCs cannot be said to be of different “types” on the basis of lineage fate outcomes [HBEH88, TSC90, WF88]. While some specified progenitors have propensities to give rise to similar cell types, these relations appear to be species-specific, and do not seem to define separate progenitor pools [AR08]. In general, then, RPCs are taken to be totipotent with respect to the neural retina- all of the cell types⁸ of the differentiated retina are derived from similar RPC lineages. Very little about this picture has changed since its initial development, using a variety of lineage tracers (including retroviruses, thymidine analogues, and injectable dyes) and histochemical markers to supplement morphological identification of specified neurons. In particular, sophisticated modern live imaging experiments in zebrafish

⁶Since both early central and later peripheral proliferation and specification are similarly, and non-trivially, organised both in space and time, this idea will be addressed in somewhat more detail below.

⁷This presumption is demonstrably incorrect in the /textit{D. rerio} eye, see Figure 2.7

⁸The “cell type” concept is unusually well-defined in the retina, as there are an abundance of distinct morphological and molecular features which differentiate numerous subtypes of the seven general types of retinal neuron.

(many pioneered by Harris), have broadly confirmed the findings of the 80s and 90s in mammalian fixed specimens, explants, and the like [BRD⁺15].

Of particular note here are the observations of the Raff group [WR90, CBR03], who demonstrated that dissociated rat RPCs, cultured at clonal density, took on morphological and histochemical features associated with different specified neural types in similar numbers and proportions, and on a similar schedule, to same-aged RPCs cultured in retinal explants. These results dramatically suggested that both the proliferative and specificative behaviour of RPCs depended less on intercellular contact, and the complex signalling environment of the developing retina, than on factors intrinsic to the RPCs themselves. These studies contain the essential germ of Harris' eventual commitment to SSM explanations, purporting as they do to “test the relative importance of cell-intrinsic mechanisms and extracellular signals in cell fate choice” and providing convincing evidence for the preponderant importance of the former.

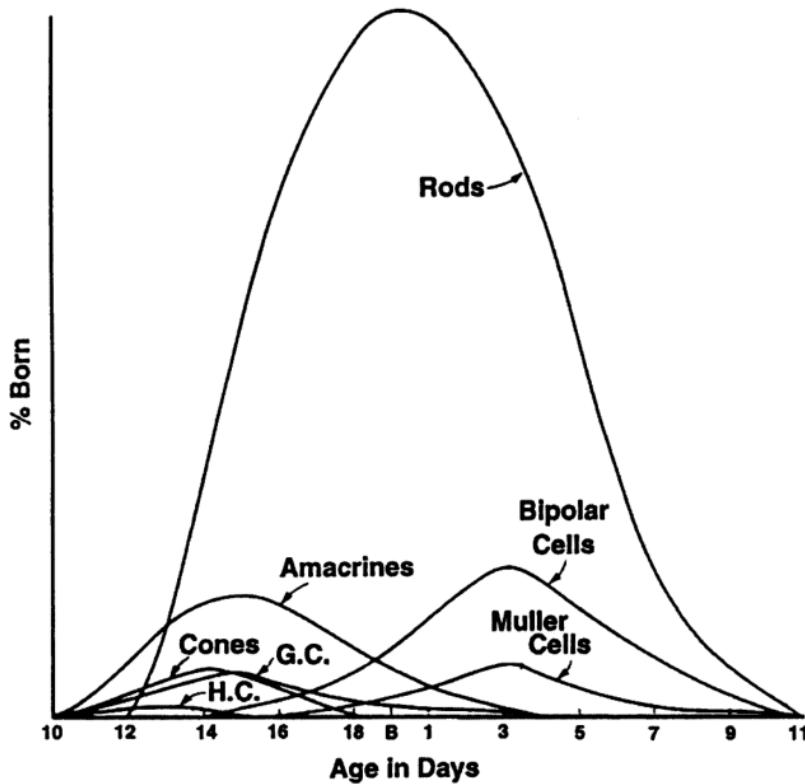


Figure 1.1: Histogenetic birth order of retinal neurons

Adapted from [You85] by [CAY⁺96]. G.C., Ganglion Cells; H.C., Horizontal Cells. Data from embryonic and perinatal mouse eyes.

In spite of the apparent ability of RPCs to produce offspring specified to any of the possible neural cell fates, at any time during their lineage history⁹, vertebrate retinal development displays a temporal ordering, such that in any particular location, retinal ganglion cells (RGCs) tend to be produced first,

⁹Even in papers arguing for a strict, linear sequence of specificative outcomes in all RPC lineages, the actual data show RPCs occasionally giving rise to “late-born” photoreceptors subsequent to their first division[WR09], giving lie to the notion that the process is particularly strict.

followed by the other cell types in what has been described as an overlapping “histogenetic order”, pictured in Figure 1.1. As noted above, RPCs also exit the cell cycle in a spatiotemporally defined order (from central to peripheral over time), and specification follows the same pattern¹⁰. This naturally gave rise to some question about the origin of the “overlap” observed in the sequential production of various cell types; conceptually, this overlap could be produced by identical RPCs executing identical rigid specifiable programs if they begin to execute it along the spatiotemporal gradient noted above, as originally suggested by Cepko et al. [CAY⁺96]. However, it is impossible to reconcile this notion with the recent results of Harris’ *in vivo* zebrafish lineage tracing studies [DPCH03, HZA⁺12, BRD⁺15], which confirm mammalian cell culture work in demonstrating that vertebrate RPC lineages do not execute identical (or even vaguely similar) specifiable programs.

1.3.3 Other morphogenetic phenomena

The outcomes of RPC proliferation and lineage commitment have generally been taken to be sufficient to explain the formation of the neural retina. Indeed, after the eye cup has been formed (prior to the specification of any neurons), RPCs are, in effect, already “in place”. Therefore, there are few documented RPC phenomena outside of the “proliferation” and “specification” categories of the morphogenetic alphabet, which enumerates the cellular processes contributing to tissue form and structure. RPCs do not seem inclined to migrate long distances, they seem not to generate much in the way of extracellular material¹¹, and, in non-pathological conditions, they are rarely seen to die. Still, there are a number of phenomena which appear to be important for proper retinal organisation that fall into these other “alphabet” categories.

The most notable of these is interkinetic nuclear migration (INM), in which RPC nuclei move back and forth between the apical and basal surfaces of the retina. Common in many neural tissues, INM affects both proliferative and differentiative outcomes, but is dissociable from them- both cell cycle and differentiation proceed (albeit in a precocious manner) when INM is disrupted[MZLF02]. The environment provided by the apical retinal surface appears to be required for RPC mitosis to proceed¹², and INM seems to consist of a directed, probably actomyosin-mediated movement to the apical surface, followed by an undirected “random walk” away from the apical surface and hence toward the basal retina [NYLH09]. This “random walk” may arise from displacements due to the directed INM of neighbouring progenitors [AHW⁺20]. More committed RPCs also appear to actively migrate to positions appropriate for the specified cell type [CAR⁺15, IKRN16], although it is unclear to what extent this short-range migration is related to INM undergone by more actively proliferating progenitors. It seems likely that these short-range migrations of more specified cells are especially significant in the early retina, before the neural plexiform layers (consisting of axons and other neural processes) have begun to divide the cellular layers into bounded compartments.

Notable lacunae in the study of the RPC morphogenetic alphabet include the regulation of cell size

¹⁰In many studies, cell cycle exit is conflated with specification such that evidence of the former is taken as evidence of the latter. There are, however, reasons to believe cell cycle exit and specification are not the same process, discussed below.

¹¹The formation of a laminin-rich basement membrane seems to be necessary for optic vesicle formation [ICW13], but it is not clear that RPCs produce this. ECM function in eye formation remains under-studied.

¹²Nonapical divisions do occur, notably in specified, but proliferative cells [GWC⁺07]. The extent to which RPCs depend on the apical surface may depend on how “RPC” is defined. In general, RPCs are no longer considered as such when they acquire characters associated with differentiated neurons (cell type markers, morphological traits, etc.), although it is clear that the acquisition of these characters does not necessarily imply that the cell is postmitotic. Any complete explanation of how RPCs give rise to the structure of the eye must also consider these nonapical divisions.

and growth, and potential roles for cell death, both of which have hardly been studied at all. While cell size and growth are known to be tightly linked to proliferative behaviour in yeast [YDH⁺11], any related effects in RPCs have not been elucidated. This may be a significant oversight, given the requirement for RPCs to continuously grow during their proliferative lifespan. Cell death does not seem to play the same “pruning” role for RPCs that it often does in other neural tissues, and observed rates of cell death in normal RPC populations are very low, so few studies have been conducted.

1.4 Macromolecular mechanistic explanations for RPC phenomena

Having surveyed the cellular phenomena pertaining to RPC function in retinal development, let us examine a selection of the many macromolecular mechanistic explanations (MEx) that have been offered to explain aspects of RPC behaviour. Unsurprisingly, given the field’s focus on the early events of eye development, these MEx are intended mainly to explain the phenomena associated with this period. Thus, the majority of MEx offered have been concerned to explain tissue-level phenomena like the initial “wave” of cell cycle exit and specification, without necessarily seeking a global explanation for RPC behaviour irrespective of context, so that it is unknown how many of these might pertain to ongoing peripheral neurogenesis or other, adult neurogenic phenomena like those exhibited by Müller glia. That said, we now turn to examine some of the best-developed of these explanations.

1.4.1 Transcription factor networks

Perhaps the most notable MEx offered to explain RPC specification and development is the eye field transcription factor network, or EFTFN. The roots of this MEx are found in the Pax6 “master gene” explanation popularised in the 1990s [Geh96]. This explanation revolved around the observation of the apparently universal involvement of Pax6 gene products in eye formation in model organisms, and the promiscuous inter-species effects of Pax6 (with mouse RNA able to induce ectopic formation of eye structures in *Drosophila* imaginal discs, for instance [HCG95]), so that it appeared to be a highly conserved genetic “switch” for eye development.

Importantly, this explanation purported to resolve what Darwin regarded as a serious problem for his theory, the apparent implausibility of the gradual evolution of eyes (and other “organs of extreme perfection”) from some primitive ancestral structure [Dar88, p.143-4]. In particular, Pax6 suggested to some theorists an alternative to the surprising suggestion of Mayr and Salvini-Plawen, that differences in eye structure and function across clades suggested the independent appearance of eyes in more than 40 clades [vM77]. Pax6 was thus taken to provide a molecular pointer to a potential common ancestor for all animal eyes [EHML09].

Subsequent investigations revealed that vertebrate Pax6 is a conserved member of a complex network of cross-activating and inhibiting transcription factors, including Pax6, Rx1, Six3, Six6, Lhx2, ET, and Tll [Zub03]. Members of this network tend to promote proliferation and suppress markers of differentiated neurons, and their loss commonly results in the failure to form the eye field at all [AH09]. The expansion of this explanation to include other TFs in a network revealed significant differences between species [Wag07]- while the role of Pax6 seems to be conserved between *Drosophila* and vertebrates, the roles of other members of the EFTFN are not. Moreover, the universality of Pax6 was only apparent, and

not real, as there are bilaterian eyes whose development is Pax6 independent (including in *Platynereis*, *Branchiostoma*, and planarians) [Koz08]. This highlighted the great difficulty in connecting morphological characters such as those observed by Mayr with a genetic basis- it is simply not clear that Pax6 conservation points to a common ancestor for all eyes, or even all photoreceptive neurons¹³. Moreover, the expansion of the moncausal “master gene” explanation, to include a network of TFs with broad gene regulatory effects, clearly highlighted the problems of complexity in offering MEx for RPC function. The components of this network interact in complex, context-dependent ways, and while the EFTFN as a whole is taken to promote RPC proliferation and to delay specification¹⁴, its individual components have also been held responsible for the specification of particular classes of differentiated neurons, and remain expressed in those postmitotic cells. Notably, Pax6 is implicated in the expression of bHLH TFs required to specify multiple classes of retinal neuron [MAA⁺01], and is known to directly activate Ath5, necessary for RGC specification [WSP⁺09]. The EFTFN has thus been taken as an explanation for the maintenance of the multipotent, proliferative RPC state, but how this network is disassembled, and its components repurposed to promote specification, remains obscure.

The EFTFN is not the only transcription factor network offered as a MEx for RPC function. Another well-developed MEx involves Chx10 (aka vsx2), a transcription factor which was found to be important for normal proliferation of RPCs, its loss causing microphthalmia in the mouse [BNL⁺96]. Chx10 was subsequently found to repress Mitf, which is involved in RPE specification, and hence to promote neural retinal fates over pigmented epithelial ones [Hor04]; in the absence of Chx10 the early eye cup does not stratify properly between apical pigmented cells and the neural retina. Much like the multifunctional EFTFN components, Chx10/vsx2 has also been implicated in the specification of particular neural fates, notably bipolar neurons [BNL⁺96] and the regulation of Vsx1 (a parologue of Chx10), Foxn4, and Ath5, associated with specification of subpopulations of bipolar cells, horizontal and amacrine cells, and RGCs and PRs, respectively [CYV⁺08, VJM⁺09].

Clear hypotheses advocating for particular relationships between different TF MEx are rarely stated. It is tempting simply to arrange them in some kind of “developmental order”, perhaps with the Chx10-Mitf network “downstream” of the EFTFN. That this would be facile is evident from the changing roles of these transcription factors depending on developmental and cellular context. To date, no unifying framework has been applied. Obvious candidates include the “developmental gene regulatory network” concept [LD09], a type of cybernetic explanation which assembles genes into feedback networks. Given the popularity of this type of explanation, it is worth noting that no one has yet had any success in offering one for RPC function.

These transcription factor network MEx frequently incorporate extracellular signals (often as an explanation for the appearance or “set-up” of the TF network), and it is generally recognised that these signals have a profound influence on these networks, and on RPC behaviour generally. We therefore turn to explanations invoking these signalling mechanisms.

¹³Indeed, the relevant “unit” of homology for evolutionary explanations for eyes remains contested, with some arguing for the cell itself over any particular set of gene sequences [EHML09]

¹⁴This is sometimes referred to as “promoting RPC fate”, since RPCs are taken to be those cells which proliferate but do not yet display markers of specification. Since it is, by now, widely recognised that cells that appear to be well-specified may remain in cell cycle [GWC⁺07, ESY⁺17] this terminology should probably be jettisoned.

1.4.2 Intercellular signalling networks

Virtually every developmentally significant class of signal has been implicated in RPC function, so that Harris, by 2009, simply glosses over a majority of these pathways by briefly summarising them in tabular form and not otherwise mentioning them [AH09]. These include BMP, CNTF, FGF, Glucagon, Hedgehog, IGF, Notch, TGF α , TGF β , VEGF, wnt, and a host of neurotransmitters. This diversity of signalling pathways has proved to be a formidable problem for integrated explanations, since almost all of these pathways converge on the same two cellular outcomes in RPCs, that is, proliferation and specification. Thus, most signalling MEx for RPC function elide the majority of other signals which are known, or thought, to affect the same processes. That said, let us explore a few of the more detailed signalling explanations.

In developmental terms, the first phenomenon requiring explanation is the appearance of the eye field to begin with- what is it that accounts for the differentiation of RPCs from the rest of the anterior neural plate and tube? Wnt signalling MEx have been offered to explain the appearance of the *Xenopus* eye field. Fz3 signalling seems to promote expression of eye field transcription factors (see below)[RDT $^{+}01$], while an unspecified non-canonical interaction between Wnt11 and Fz5 inhibits canonical β -catenin signalling through Wnt8b/Fz8a, which would otherwise promote prospective anterior forebrain fates [CCY $^{+}05$]. Inhibition of FGFR2 signalling, and activation of ephrinB1 signalling have also been implicated in early *Xenopus* eye field cell movements [MMDM04]. Subsequent experiments determined that the xenopus ADP signalling through the P2Y1 receptor directly activates the eye field transcription factor network (EFTFN), another well developed MEx described below [MBE $^{+}07$]. More recent experiments in zebrafish suggest that precocious acquisition of neuroepithelial apicobasal polarity, probably driven by interactions with a Laminin1 basement membrane, distinguishes the early eye field [ICW13].

Developmentally subsequent to the appearance of eye field RPCs and their rearrangement into the optic cup, the apparent central-to-peripheral “wave” of RPC exit from cell cycle and specification of early RGCs [HE99], has had detailed MEx advanced to explain it. In both zebrafish and chick retina, FGF3 and FGF8, originating from the optic stalk, initiate this early cell cycle exit and specification [MDBN $^{+}05$], while inhibiting FGF signalling prevents this from occurring, and ectopic expression of FGF can cause it to occur inappropriately. The progression of this “wave” of cell cycle exit and specification has been separately explained, by Sonic Hedgehog (Shh) signalling from the newly specified RPCs inducing cell cycle exit and specification in adjacent cells [Neu00]. This process is dependent on, and downstream of, the above-mentioned FGF induction [MDBN $^{+}05$]. The role of Hh signalling has been challenged on the basis that Hh inhibition in subsequent experiments did not display the same effect size [SF03], and that its effects on Ath5 expression, required for RGC specification, are ambiguous [AH09]. More recent MEx advanced by Harris have suggested that Hh signals may decrease the length of the cell cycle, resulting in increased proliferation and earlier cell cycle exit and specification [LAA $^{+}06$, ALHP07].

A well-developed “local” signalling MEx (mainly advanced by Harris) invokes the classic Notch/Delta lateral inhibition model, with small fluctuations in Notch/Delta activity giving rise to a positive feedback response that differentiates neighbouring cells. Cells which have high Delta expression tend to be specified as retinal neurons, while those with high Notch tend to remain proliferative, either as RPCs or Müller glia [DRH95, DCRH97]. Such a mechanism could regulate the activity of both early, central RPCs, as well as peripheral RPCs, and may contribute to inter-RPC variability. These differences between RPCs located in different parts of the developing retina have been of significant interest, and it is worth briefly examining patterning MEx that may also explain spatial differentiation between RPCs.

1.4.3 Patterning mechanisms

Among the most interesting features of RPCs is that they reliably give rise to specified neurons, in particular RGCs, that seem to “know where they are” in the retina, enabling them to wire their axons in correct retinotopic order in the superior colliculus (SC) or optic tectum (OT). The most robust MEx explaining this refer to gradients of EphA and EphB receptors expressed in RGCs, and their respective ephrin ligands expressed in the SC or OT. In the retinal RGC population, an increasing nasotemporal gradient of EphA is paired with an increasing dorsoventral gradient of EphB. A corresponding increasing rostrocaudal gradient of ephrin-A is paired with an increasing lateromedial gradient of ephrin B in the SC/OT. This allows for a two-axis encoding of an RGCs’ position in the retina [TK06]. As the RGCs’ axon pathfinding depends on repulsive effects mediated by Eph receptors, this code is sufficient to allow correct wiring of even single RGCs [GNB08]. This code seems to be established, in part by the action of Gdf6a, in RPCs themselves, prior to specification [FEF⁺09].

Indeed, there are numerous similar observations of expression gradients that create spatial differences between RPCs themselves. Most relevant to the proliferation dynamics highlighted in this chapter is the observation that, in *Xenopus* eyes, a decreasing dorsoventral gradient of type III deiodinase renders the cells of the dorsal CMZ refractory to thyroid hormone (as the deiodinase inactivates TH) [MHR⁺99]. The effect of this is to set up a differential response to TH in post-metamorphic RPCs, so that the ventral population selectively expands in response to TH [BJ79].

These patterning mechanisms are of particular interest here, in large part because they clearly establish that the RPC population is heterogenous, both with respect to proliferative and specificative behaviours, and perhaps others as well. This is of critical importance for any modelling effort, as virtually all mathematical models used by stem cell biologists (and those used to justify Harris’ SMME) assume, at least initially, homogenous populations of stem or progenitor cells. Since RPCs do not meet this condition, special care is needed to use these models.

1.4.4 Chromatin dynamics

In recent years, the great importance of chromatin conformation in RPC proliferation and specification has become more clear. Indeed, chromatin dynamics are now widely invoked in explaining stem and progenitor cell behaviour, and suggested as a target for cell reprogramming [Kon06, TR14]. In RPCs, detailed accounts of three-dimensional chromatin dynamics have yet to appear. However, a number of studies point to the importance of chromatin state in informing the overall cellular state. In particular, histone deacetylation seems to be important for RPC specification, as the loss of histone deacetylase 1 (HDAC1) in zebrafish results in overproliferation and decreased specification, and correlated increases in Wnt and Notch activity [Yam05]. In mouse retinal explants, pharmacological inhibition of HDAC results in decreased proliferation and specification [CC07]. Additionally, the chromatin remodelling complex SWI/SNF has repeatedly been implicated in RPC function. Notably, one particular component of this complex seems to be particularly associated with vertebrate RPCs (BAF60c, an accessory subunit) [LHKR08]. A switch to other subunits seems to be necessary for specification [LWR⁺07]. Details regarding the subunits involved in specification and their downstream effects are complex and context dependent, much like the signalling pathways mentioned above.

1.5 A unified theory of RPC function? “Blurring” to order

From the foregoing discussion, we can clearly see Harris’ theoretical conundrum. Macromolecular explanations for RPC behaviours, like those throughout the molecular biological tradition, have generally been built outwards from particular transcription factors, receptors, etc. The result is an archipelago of MEx, at best connected by tenuous speculation, and in most cases, without any known means to form an integrated model. Furthermore, the degree of complexity and context-dependence evident from the literature might seem to preclude such a model. As we have seen, Harris found that the evidence did not allow for clear discrimination between logically distinct types of mechanisms for producing the observed variability in RPC lineage outcomes.

In this situation, Harris effectively had two theoretical options. The first is simply to “crank the handle”- to generate more and more facts describing the difference particular molecules make to RPC outcomes in dozens of relevant contexts, piling up exceptions and idiosyncrasies, in the hope that doing so will eventually bridge the explanatory “islands” of the MEx archipelago. I have referred to this as the Enumerationist approach and explained why it has failed, and continues to fail, in the Theoretical Appendix.

The second option, the one actually chosen by Harris, is more theoretically sophisticated. As Nicholas Rescher has noted regarding the in-principle limits to scientific knowledge, the phenomenal universe has infinite descriptive complexity- one can always add more detail to a description of some phenomenon, and no such description is ever complete [Res00, p.22-9]. Moreover, “even as the introduction of greater detail can dissolve order, so the neglect of detail can generate it.” [Res00, p.62] As Rescher goes on to comment:

[W]e realize that in making the shift to greater detail we may well lose information that was, in its own way, adequate enough ... information at the grosser level may well be lost when we shift to the more sophisticated level of greater fine-grained detail. The ‘advance’ achieved in the wake of ‘superior’ knowledge can be - and often is - purchased only at a substantial cognitive loss.

...

It is tempting on first thought to accept the idea that we secure more - and indeed more useful and more reliable - information by examining matters in greater precision and detail. And this is often so. But the reality is that this is not necessarily the case. It is entirely possible that the sort of information we need or want is available at our ‘natural’ level of operation but comes to be dissolved in the wake of greater sophistication. [Res00, p.65-6]

Rescher’s greater point is that “blurring” detail, at levels below the phenomenal one under consideration (for RPCs, generally, the cell or lineage), may actually be necessary to produce an ordered explanation that is useful with respect to some objective. Given the number of apparently contradictory MEx for RPC behaviour, we have exactly the kind of situation Rescher is describing- more sophistication, and more detail, has dissolved order, not revealed it¹⁵. The inability to assemble an unified explanation for RPC function has left us without a good fundamental understanding of how highly ordered neural tissues like eyes can be generated from composites of units with highly variable, temporally and spatially ordered outcomes like RPC lineages. As this is a common feature of vertebrate neurogenesis

¹⁵ At least part of this problem is likely related to the fact that the majority of biomedical findings cannot be replicated [Ioa05]. The finding, mentioned above, that Shh effect sizes on RPC function were not as large as initially reported when subsequently investigated, is typical and symptomatic of this replication problem. “Blurring” may therefore be necessary not only because of fundamental epistemic limits, but also because it is often difficult to distinguish bona fide results and explanations from spurious ones.

more generally, the theoretical problem here leaves us without the ability to produce complete models of neurodevelopmental processes in many species. Moreover, in the absence of a clear framework for comparing the explanatory power of the diverse array of MEx so far advanced, practical contributions of clinical relevance have been scanty and tentative, with RPC transplantation, and more recently, gene therapies taking little note of complex MEx for RPC function[CAI⁺04, GS07, YQW⁺18].

In a situation of this kind, it may well be that this type of “blurring” is required, and it seems that is what Harris is attempting in advancing his SMME. Harris is asking, in some sense, whether most of the MEx offered for RPC behaviour are extraneous to an adequate understanding of how RPCs work. By cutting down to the simplest possible explanation, Harris hopes to bring into view order that was previously obscured by detail. There is, of course, a significant danger here: how does one decide what is “blurred out” and what remains? We can easily understand how a practitioner’s biases could lead to a sort of relativism, where the “blurring” makes apparent a spurious order that conforms to these biases rather than to reality as such. With this in mind, let us survey the general thrust of Harris’ SMME, before proceeding to examine it in detail, in Chapter 2.

1.6 Explanatory Strategy and Intent of the SMME

As we have seen in Section 1.2, Harris’ long-held understanding of the explanatory options for RPC function divides them into four broad categories, which I summarise as follows:

1. A linear algorithmic “program” of proliferation and specification
2. Asymmetric segregation of specificative determinants
3. “Stochastic processes” internal to the cells
4. Influences of extracellular factors

Harris’ sophisticated discussions of RPC MEx rarely treat these categories as exclusive, and concede that good explanations for RPC behaviour may involve phenomena from more than one of them. Indeed, the SMME necessarily contains elements that Harris concedes are “linear” and “deterministic” [HZA⁺12]. Still, his overall strategy for the SMME is, first, to substantiate the predominant influence of one of these categories of phenomena (that is, category 3, internal stochastic processes or effects), and subsequently to specify an actual macromolecular system that could plausibly be such an “internal stochastic process”. These two theoretical maneuvers, while tightly linked, serve different purposes within Harris’ overall explanatory framework, which must be examined separately.

The SMME for zebrafish RPC function has been developed across three separate papers [HZA⁺12, BRD⁺15, WAR⁺16]. Each builds on the earlier publications, collectively purporting to explain the behaviour of RPCs wherever, and whenever, they may be found in the zebrafish eye. The underlying model is originally derived from an earlier paper pertaining to rat RPCs [GZC⁺11]. He et al. [HZA⁺12] and Wan et al. [WAR⁺16] use essentially the same model and make up the substance of the first of these two maneuvers. Boije et al. [BRD⁺15] substantially modifies this model, specifying the activity of two known transcription factors (Ath5 and Ptf1a) as the model’s biological referents. This paper constitutes the second theoretical thrust.

The first maneuver intends to provide support for the contention that zebrafish RPCs are a group of equipotent cells which give rise to variable outcomes that depend on independent “stochastic” processes

within each of these cells. This support is to be provided by demonstrating that a suitably configured Simple Stochastic Model (SSM) of an RPC, numerically simulated many times by Monte Carlo methods to represent a population of RPCs, produces similar outcomes to populations of RPCs *in vivo*. This explanatory strategy is common in the stem cell literature, the original example having been published in 1964 by Till, McCulloch, and Siminovitch[TMS64]. The SMME therefore represents an example of a traditional scientific logic- an explanatory pattern deployed by stem cell biologists in diverse contexts, and widely accepted because of its ongoing use in the literature, although with varying interpretations.

The success of this first maneuver thus depends on two outcomes. Firstly, the output of the SSM should accurately reflect the observed proliferative and specificative outcomes of zebrafish RPC lineages, giving weight to Harris' claim that it "provides a complete quantitative description of the generation of a CNS structure in a vertebrate *in vivo*" [HZA⁺12]¹⁶ Secondly, the internal structure of the SSM should provide good reason to believe that one of the Noise explanations is a better explanatory option for RPC lineage outcomes than those identified by the other three categories of theoretical options enumerated above.

The second theoretical maneuver is the specification of particular biological referents for entities in the model. This offers an opportunity to move beyond a purely conceptual argument about the kind of process that might produce variable RPC lineage outcomes, and to begin the work of explaining how the behaviour of a particular macromolecular system constitutes such a process, so that empirically verifiable hypotheses may be generated. The success of this maneuver depends on the biological plausibility of the identification between model structure and the biological function of transcription factors, *Ath5* and *Ptf1a*, that the model names. A good SSM-based explanation would point the way for further research by identifying *how* so-called "stochastic processes" in RPCs might function, and make some predictions about this. With that said, let us turn to the SMME and determine how well these manuevers have succeeded.

¹⁶This claim is somewhat extravagant; the SSM, by definition, includes no spatial information, so it is unclear how one could be a "complete description" of any spatially organised structure. Still, it can be complete with regard to cell population numbers.

Chapter 2

“Stochastic mitotic mode” models do not explain zebrafish retinal progenitor lineage outcomes

The text of this chapter has been adapted from a manuscript originally prepared for submission to PLOS One and is formatted accordingly; the methods and supplementary materials are available in ??

2.1 Introduction

Mechanistic explanations (MEx) derive their utility from the resemblance of the conceptual mechanism’s output to empirically observed outcomes. As maps to biological territories, biological MEx are naturally identified with the living systems they represent. Most well-developed MEx take the form of a model, whose internal structure is taken to reflect the underlying causal structure of a biological phenomenon. The nature of the causal relationship between a mechanistic model and the phenomenon it purports to explain remains a topic of active dispute in the philosophy of biology[Fag15]. Biologists, nonetheless, usually accept that a model which explains empirical observations well (usually measured by statistical or information theoretic methods), and reliably predicts the results of interventions, bears a meaningful structural resemblance to the actual causal process giving rise to the modelled phenomenon.

Biological phenomena are notable for exhibiting both complex order and unpredictable variability. A significant challenge for MEx in multicellular systems is to explain how complex, highly ordered tissues, like those produced by neural progenitors, can arise from cellular behaviours with unpredictably variable outcomes. Stem cell biologists have traditionally resorted to Simple Stochastic Models (SSMs) in order to explain the observed unpredictable variability in clonal outcomes of putative stem cells[TMS64, Fag13]. Because SSMs are susceptible to Monte Carlo numerical analysis as Galton-Watson branching processes, they have been convenient explanatory devices, appearing in the literature for more than half a century. By specifying the probability distributions of symmetric proliferative (PP), symmetric postmitotic (DD), and asymmetric proliferative/postmitotic (PD) mitotic modes, SSMs allow cell lineage outcomes to be simulated.

SSMs are “stochastic” insofar as they incorporate random variables with defined probability distribu-

tions. As Jaynes has noted, “[b]elief … that the property of being ‘stochastic’ rather than ‘deterministic’ is a real physical property of a process, that exists independently of human information, is [an] example of the mind projection fallacy: attributing one’s own ignorance to Nature instead.”[JBE03] Despite this, macromolecular processes are often described as “stochastic” in the stem cell literature. Generally speaking, the behaviour of an SSM’s random variable is taken to represent sequences of outcomes that are produced by multiple, causally independent events. Recently, the influence of “transcriptional noise” on progenitor specification has been identified as a candidate macromolecular process that may produce unpredictable variability in cellular fate specification. Therefore, one explanatory strategy for stem and progenitor cell function compares SSM model output to observed lineage outcomes, in order to argue that “noisy”, causally independent events give rise to the proliferative and speciative outcomes of progenitor lineages.

In this report, we evaluate one of the best-developed of these explanations, proffered by William Harris’ retinal biology group. We have dubbed this the ”Stochastic Mitotic Mode Explanation” (SMME) for zebrafish retinal progenitor cell (RPC) function. The SMME is noteworthy because it claims: (1) to ”provid[e] a complete quantitative description of the generation of a CNS structure in a vertebrate in vivo”[HZA⁺12]; (2) to have established the functional equivalency of embryonic RPCs and their descendants in the postembryonic circumferential marginal zone (CMZ)[WAR⁺16]; and (3) to have established the involvement of causally independent transcription factor signals in the production of unpredictably variable RPC lineage outcomes[BRD⁺15]. Moreover, the SMME is taken to explain histogenetic ordering (the specification of some retinal neural types before others, notably the early appearance of retinal ganglion cells (RGCs)) in vertebrates, without reference to the classical explanation of a temporal succession of competency states[TR86]. These would be significant achievements with important consequences for both our fundamental understanding of CNS tissue morphogenesis and for retinal regenerative medicine. However, these models were not subjected to typical model selection procedures, nor has their output been examined using modern information theoretic methods, as advocated by mainstream model selection theorists[BA02].

In order to facilitate the critical evaluation of the two SSMs which form the SMME’s MEx for RPC function, we have re-expressed the models as cellular agent simulations conducted using the open source C++-based CHASTE cell simulation framework[MAB⁺13]. We explored the structure of the SSMs, dubbed the He and Boije SSM respectively, compared to their explanatory forebear, dubbed the Gomes SSM[GZC⁺11]. This analysis strongly suggested that the introduction of an unexplained progression of temporal “phases” was largely responsible for the SMME model fits to data. In order to investigate this possibility, we built an alternative model with a deterministic mitotic mode and compared its output to the He SSM. We found that the deterministic alternative model was a better explanation for the observations, demonstrating that SMME fails when compared to alternatives. We therefore suggest that an explanatory approach based on the use of SSMs is incapable of distinguishing between theoretical alternatives for the causal structure of RPC lineage behaviours. Furthermore, by comparing the output of the models with novel postembryonic measurements of proliferative activity, we find that the SMME explanation cannot account for quantitative majority of retinal growth in the zebrafish, driven by the CMZ. Finally, we discuss the place of SSMs, the concept of “mitotic mode”, and the role of “noise” in explaining RPC behaviour, and suggest ways to avoid the modelling pitfalls exemplified by the SMME.

2.2 Results and Discussion

The SMME for zebrafish RPC function has been advanced using two SSMs. One first appears in He et al., and again, unmodified, in Wan et al.[HZA⁺12, WAR⁺16]; it is concerned primarily to explain lineage population statistics and time-dependent rates of the three generically construed mitotic modes (that is, PP, PD, DD). We have called this the He SSM. The second appears in Boije et al.[BRD⁺15]; its intent is both to introduce the role of specified macromolecules into the mitotic mode process, and to explain neuronal fate specification in terms of the process. We have called this the Boije SSM. The He model is directly descended an SSM advanced to investigate causally independent fate specification in late embryonic rat RPCs, formulated in Gomes et al.[GZC⁺11]. The Boije SSM differs substantially from the He and Gomes models, but inherits its general structure from the He SSM.

The metascientific analysis of biological explanations developing in time remains understudied. Perhaps most refined tool for global evaluation of biological theories (Schaffner's "Extended Theories"), treats biological explanations as hierarchically organised logical structures, after the fashion of Imre Lakatos, and proposes the use of Bayesian logic to distinguish between them[Sch93]. However, biologists rarely offer explanations in this form; we rather prefer MEx, expressed in diagrammatic form or as mathematical model-objects.

In this report, we accept Fagan's view that MEx for the behaviour of stem and progenitor cells consist of assemblages ("mechanisms") of explanatory components which are understood to be causally organised by virtue of their intermeshing properties[Fag15]. While Fagan treats SSMs separately from macromolecular MEx, and we find that the Gomes SSM was not deployed in this role, the He and Boije SSMs were used as explanations for the behaviour of RPCs. As Feyerabend famously observed, scientists often operate as epistemological anarchists; the development of our explanatory logic is not bound by an identifiable set of rules, but rather arises organically from our scientific objectives, extrascientific context, and so on[Fey93].

We have therefore chosen to examine the structure of the Gomes, He, and Boije SSMs arranged in chronological order, to highlight how the explanatory logic of zebrafish SMME SSMs differs from their immediate ancestor, and from the traditional uses of SSMs in stem cell biology. We have diagrammatically presented these SSMs as MEx, consisting of components describing the proliferative and fate specification behaviour of cellular agents. The proliferative and specificative components of the MEx are causally organised by their Faganian intermeshing property, mitotic mode. We have used abbreviations to denote important classes of model components and inputs, informed by the emphasis of Feyerabend on the persuasive role of metaphysical ingredients and auxiliary scientific material; these are as follows:

- MI - Model ingredient, making reference to some conceptual or metaphysical construct
- AS - Auxiliary scientific content
- RV - Random variable
- PM - Parameter measurement, model parameter set by measurements, independently of model output considerations
- PF - Parameter fit, model parameter set without reference to measurement, in order to produce model output agreement with observations

We draw the reader’s attention to the expansion of the “mitotic mode” intermeshing property in later SSMs; this explanatory component comes to dominate the later models, which makes its interpretation critical to the success of the mechanistic explanation in meaningfully representing a real biological process.

Numerous theoretical options to explain observed variability in RPC lineage outcomes have historically been considered. Harris has previously argued that these belong to three cell-autonomous categories of process, in addition to extracellular influences: (1) a linear temporal progression of competency states, (2) asymmetric segregation of determinants during mitoses, and (3) intracellular “stochastic events”[HBEH88, AH09]. All of the SSMs are used to argue for the predominant influence of the third type of process in RPC lineage outcomes, essentially by demonstrating that the output of the SSM resembles observations.

2.2.1 Gomes SSM: Ancestral Model of the SMME SSMs

The Gomes SSM is presented in Fig 2.1. The model’s structure is straightforward; there are three independent random variables, drawing from empirically-derived probability distributions for the time each cell takes to divide, the mitotic mode of the division, and the specified neural fate of any postmitotic progeny. The random mitotic mode variable functions as the “intermeshing property” linking cycle behaviour to fate specification. The model thus represents a scenario where the processes governing proliferation, leading to cell cycle exit, and governing cell fate commitment are, at every stage, causally independent of each other and of their foregoing history of outcomes. The objective of the Gomes et al. study is to compare the lineage outcomes of dozens of individual E20 rat RPCs in clonal-density dissociated culture with the model output, developing earlier work in this system[CBR03]. Although the model incorporates conventional proper time (clock-time), none of the RVs reference it to determine their values. The abstract “cells” represented by this model do not have any timer or any source of information about their relative lineage position. Fate specification is construed in terms of conventional histochemical markers of stable cell fates; only the neural types generated late in the retinal histogenetic order are represented, as these are the only neurons specified by E20 rat RPCs.

This SSM is explicitly built as a null hypothesis, or a model of background noise. It represents an extreme case in which all of the specificative and proliferative behaviours of an RPC are totally independent of all other RPCs and events in its clonal lineage. The stated purpose of this model is “to calibrate the data”, serving “as a benchmark”[GZC⁺11], a purely hypothetical apparatus to produce sequences of causally unrelated lineage outcomes. Substantial deviations of the observed data from the fully independent events of the SSM may then be interpreted as causal dependencies between RPC outcome and their relative lineage relationships, as might be observed in a developmental “program” or algorithmic process. Finding that, with some exceptions, observed proliferative and specificative outcomes generally fall within the plausible range of Gomes SSM output, Gomes et al. conclude that RPC lineage outcomes in late embryonic rat RPCs seem to be dominated by causally independent events, suggesting that causally isolated “noisy” macromolecular processes may give rise to the unpredictable variability in the behaviour of these cells.

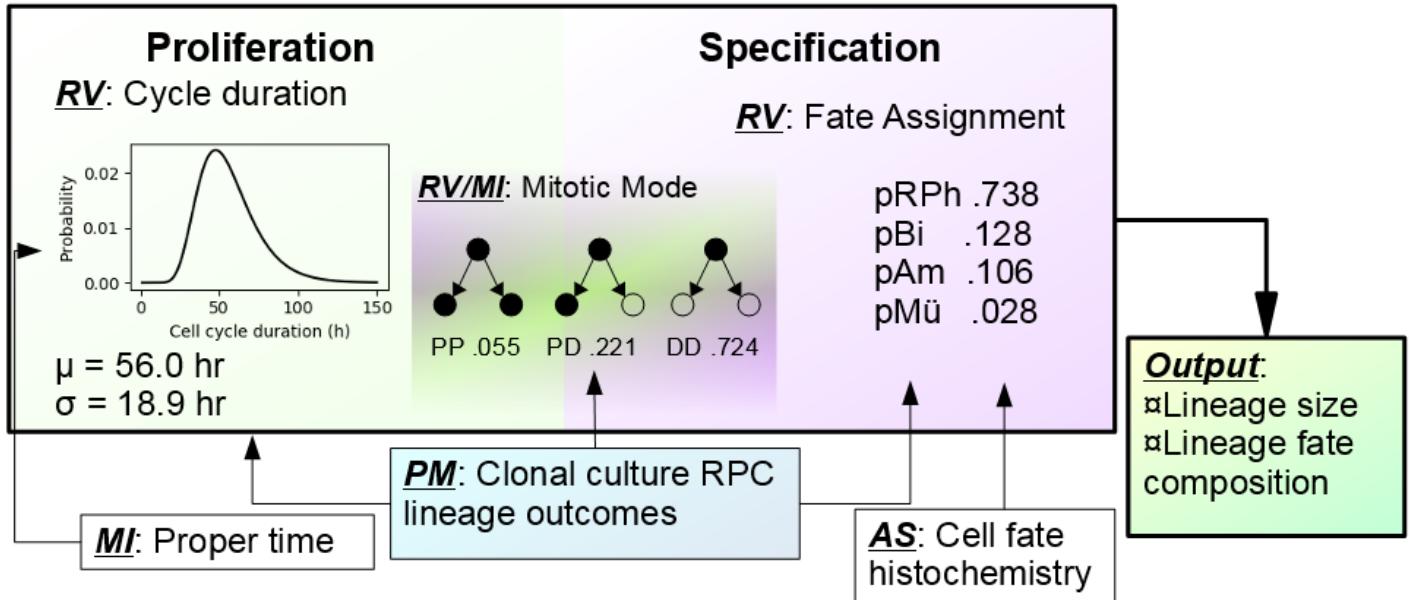


Figure 2.1: Structure of Gomes SSM

Structure of the Gomes SSM. pRPh, probability of rod photoreceptor specification. pBi, probability of bipolar cell specification. pAm, probability of amacrine cell specification. pMü, probability of Müller glia specification.

2.2.2 He SSM: Explaining variability in zebrafish neural retina lineage size

The He SSM, shown in Fig 2.2, is deployed in a explanatory role rhetorically identical to the Gomes SSM. The extent to which model output “captures.. aspects of the data” is taken to obviate the need for explicit “causative hypothes[es]”. On this account, only residual error between model output and observations may be ascribed to non-stochastic processes like “histogene[tic ordering] of cell types or a signature of early fate specification”. That is, if the model can be fit to observations, this is taken to exclude the presence of any cell-autonomous temporal program, asymmetric segregation of fate determinants, extracellular influences, and the like.

There are fewer direct empirical inputs to the He model’s parameters than the Gomes SSM, limited to the duration of the cell cycle. The close synchrony of zebrafish RPC divisions is modelled by assigning sister RPCs the same cycle length, shifted by a normal distribution of one hour width, in contrast to Gomes RPCs, which are treated as fully independent. The lineage outcomes the He SSM is called on to explain differ significantly from those referred to by the Gomes SSM. Most notably, the Gomes SSM does not account for the early appearance of RGCs, which are not produced by the late E20 progenitors examined in that study. The zebrafish RPC lineages studied by He et al. produce all of the retinal neural types, including RGCs, which are typically produced by PD-type divisions. The He SSM does not model particular cell fates, supposing that mitotic mode is “decoupled” from fate specification. The mitotic mode RV linking cell cycle to fate specification in the Gomes SSM has thus subsumed the specification outcomes of RPC lineages entirely, and the model is concerned only to explain the sizes of lineages (marked by an inducible genetic marker at various times), and the observed progression of mitotic modes in these early RPCs.

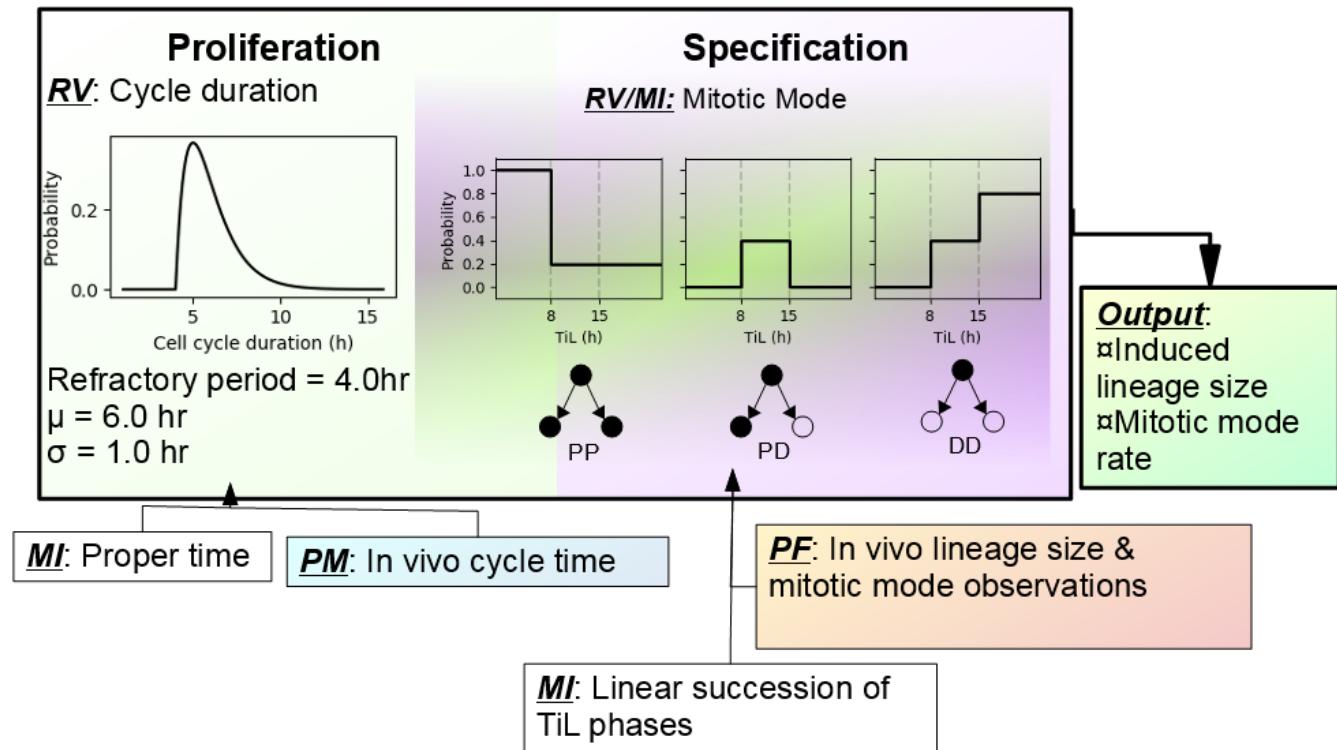


Figure 2.2: **Structure of He SSM**
Structure of the He SSM. TiL, Time in Lineage.

The He model is, therefore, called on to explain the temporal structure of the proliferative and speciative behaviour of early zebrafish RPCs that dissociated late rat RPCs do not exhibit. A Gomes-type SSM, in which the RVs determining RPC behaviour are independent of any measure of time, cannot account for this temporal progression. In order to address this, He et al., assume a linear progression of three phases which cells in each lineage pass through, the timing of these phases being determined relative to the first division of the RPC lineage, called here “Time in Lineage” or TiL. The values the mitotic mode RV may take on is determined by these TiL phases. The temporal structure of the phases and their effect on the mitotic mode RV are selected to produce a model fit. While He et al. acknowledge that the model therefore represents a “combination of stochastic and programmatic decisions taken by a population of equipotent RPCs,” no effort is made to determine the relative contribution of the model’s stochastic vs. linear programmatic elements. Instead, the purportedly stochastic nature of mitotic mode

determination is emphasized throughout the report.

2.2.3 Boije SSM: Explaining variability in zebrafish RPC fate outcomes

The second SMME model advanced to explain zebrafish RPC behaviour, the Boije SSM, is displayed in Fig 2.3. This model is primarily concerned with the lineage fate outcomes that the He SSM does not treat, while abandoning the explicit proper time of the He and Gomes SSMs in favour of abstract generation-counting. The mitotic mode model-ingredient now subsumes all RPC behaviours. Boije et al. make a laudable effort to specify the particular macromolecules ostensibly involved in determining mitotic mode, nominating the transcription factors (TFs) Atoh7, known to be involved in RGC specification, and Ptfla, known to be involved in the specification of amacrine and horizontal cells, as primary candidates. The contribution of vsx2 is taken to determine the balance between PP and DD divisions late in the lineage, with the latter resulting in the specification of bipolar or photoreceptor cells. The binary presence or absence of these signals is determined by independent RVs structured by the phase structure present in the He SSM, translated into generational time from proper time.

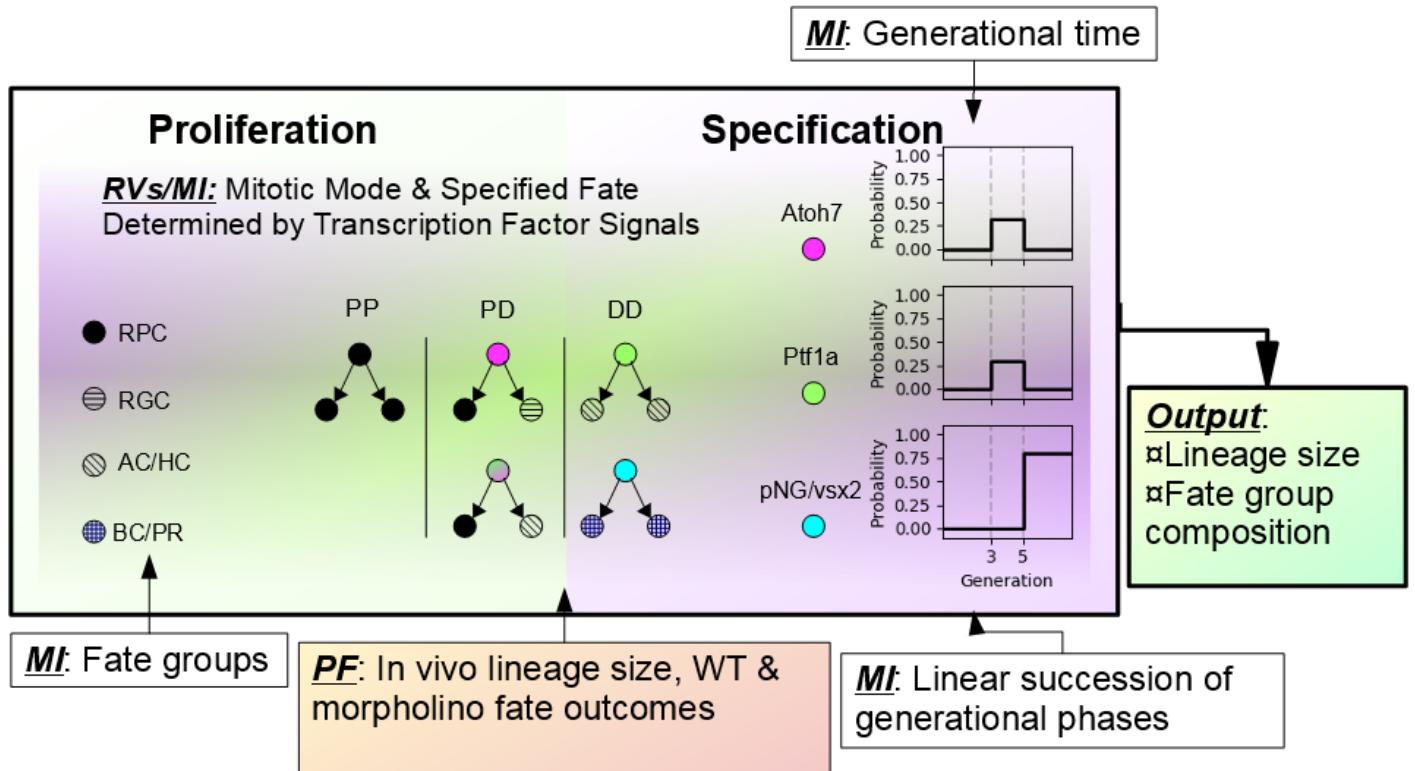


Figure 2.3: Structure of Boije SSM

Structure of the Boije SSM. RPC, retinal progenitor cell. RGC, retinal ganglion cell. AC, amacrine cell. HC, horizontal cell. BC, bipolar cell. PR, photoreceptor. pNG- parameter representing contributions of Vsx1 and Vsx2 to specification of BC & PR fates in absence of Atoh7 or Ptfla signals.

By this point, the SMME has become a very different type of explanation from its Gomes SSM forebear. We are no longer dealing with RVs that model causally and temporally independent processes

for different aspects of RPC behaviour. There is, rather, one temporally dependent process, the determination of mitotic mode, which is explained by unpredictable subsets of each lineage generation expressing particular TF signals. Where the Gomes SSM takes its parameters directly from empirical measurements of lineage outcomes, asking whether it is sufficient to assume that these are independently determined, the Boije SSM’s parameters are derived solely from model fit considerations. In spite of these considerable differences, the explanatory role of the SSM is effectively the same: the model’s fit to observations is taken as evidence of the predominant influence of “stochastic processes” determining mitotic mode on RPC behaviour. While Boije et al. acknowledge that whether some process is called “deterministic” or “stochastic” is “a matter of the level of description”[BRD⁺15] (i.e. is a property of the model-description and not of the physical process), the explanatory role of stochasticity for RPC lineage outcomes is, once again, emphasized throughout.

2.2.4 Model selection demonstrates the SMME is not the best available explanation for RPC lineage outcomes

From a modeller’s perspective, it is notable that neither of the reports which use the He SSM[HZA⁺12, WAR⁺16], nor that using the Boije SSM[BRD⁺15] report in any detail their fitting procedures, nor do any of the above report any statistical measures of goodness-of-fit. Additionally, no models representing the alternative “theoretical options” available to explain variability in RPC lineage outcomes are compared to those advanced as evidence for “stochastic processes”. Given that the He SSM and Boije SSM depart from the Gomes SSM by the addition of an unexplained temporal structure to the mitotic mode model ingredient, it is striking that the overall argument remains similar to Gomes et al.’s, despite the persuasive force of the latter deriving from the lack of such structures. While He et al. and Boije et al. acknowledge that their models involve both stochastic and linear programmatic elements, no effort is made to quantify their relative influence on model output, and macromolecular explanation is only applied to the stochastic elements. Moreover, the emphasis on this mitotic mode model construct increases with each successive model, to the extent that in the Boije model there are no other elements that are used to explain RPC behaviours. All cellular behaviours are, in effect, progressively collapsed into this stochastic mitotic mode concept. Finally, the He and Boije SSMs contain more parameters, which are determined by fewer empirical measurements than the Gomes SSM. Clearly, then, the SMME SSMs are at significant risk of representing trivial overfits to their data, the modeller’s equivalent of the “just-so” story, not representing any actually-existing macromolecular or cellular process.

Fortunately, we may employ standard statistical model optimisation and selection techniques to adjudicate this possibility. The most straightforward way to do so is to reexamine the He SSM alongside a competing alternative model. The data to which the He SSM has been applied is particularly amenable to a scheme in which the models are fit to a “training” dataset, followed by a “test” dataset. In this case, the training data are the induced lineage size and mitotic mode rate data from He et al., and the test dataset are the Atoh7 morpholino observations from He et al., and the CMZ lineage size data from Wan et al. This allows us to test how well the He SSM, and an alternative, hold up under novel experimental conditions, without relying on a trivial ordering of goodness-of-fit to training data. Since the Boije SSM draws straightforwardly on the He SSM for its temporal phase-parameterisation and stochastic mitotic mode model ingredient, this analysis of the He SSM also bears directly on the validity of the Boije SSM.

As an alternative to the SMME He SSM, we constructed a model that differs only in its construal of the process governing the RPC mitotic mode. Rather than variability arising from a stochastic-

process mitotic mode changing across phases of fixed length, we simply supposed that mitotic mode is deterministic in each phase (guaranteed PP mitoses in the first phase, PD in the second, and DD in the third), but the phase lengths are variable between lineages and shift slightly between sister cells. That is, we represented this linear progression of deterministic mitotic mode phases using the same type of statistical construct the He SSM applies to model cell cycle length, to avoid introducing any novel or contentious elements into the model comparison. More specifically, each lineage has a first PP phase length drawn from a shifted gamma distribution, followed by a second phase length drawn from a standard gamma distribution. Upon mitosis, these phase lengths are shifted in sister cells by a normally distributed time period, exactly like cell cycle lengths in the He SSM.

We take this to be a reasonable representation of the classic suggestion that RPCs step through linear succession of competency phases, given the conceptual tools that the He SSM uses to model mitotic phenomena. If we suppose that this temporal program is governed by RPC lineages passing through a stereotypical series of chromatin configurations which allow for PP, then PD, then DD mitoses in turn, along with the associated competence to produce the particular cell fates associated with PD and DD mitoses, it seems entirely plausible to suggest that lineages differ in the lengths of time they occupy each state. This is particularly true if we concede that an SSM, forego any spatial modelling whatsoever, must necessarily abstract extracellular and spatial influences on these processes. Moreover, since these chromatin configurations must be broken down and rebuilt with each mitosis, the re-use of the “sister shift” model ingredient from the He SSM’s cell cycle RV is congenial, representing the same sort of cell-to-cell variability that results in the small differences between sister cells in cycle timing.

While the code used to implement the SSMs mentioned above has not been published, the relevant reports provide enough detail to reconstruct these models in full, which we did using the CHASTE cell-based simulation framework, in order to provide transparent and reproducible implementations of the SSMs. Because the values selected for the He SSMs’ parameters seem to have been selected as a series of rough estimates, with only the value for the probability of PD-type mitoses in the second model phase being varied to produce the fit, we suspected that the fit would not be at or near the local minimum for any reasonable loss function. That is, a model fit produced in this manner is likely to be located in a region of the parameter space that is highly sensitive to small perturbations, and therefore may depend strongly on implementation-specific idiosyncrasies. He et al. report that they experienced difficulty in obtaining a good fit to their 32 hour induction data, with changes in the phase two PD probability producing large differences in fit quality. Unsurprisingly, when we rebuilt the He SSM with the original fit parameterisation, we substantially reproduced the original fit, except for the 32 hour data, where the model output diverges substantially from that reported in He et al. (see [S1 Fig](#)). Since this is clearly not the best fit available for the He SSM, and we wish to directly compare the best fits (i.e. the parameterisations at minima of some loss function) for the He SSM and our putative alternative model, we used the simultaneous perturbation stochastic approximation (SPSA) algorithm[[Spa98](#)] to optimise both model fits. SPSA is particularly convenient for complex, multi-phase models like the He SSM, because no knowledge of the relationship between the model’s parameters and the loss function is required. We used Akaike’s information criterion (AIC) as the loss function to be minimised, in order to provide a rigorous comparison between the two differently-parameterised models (the deterministic alternative has two fewer parameters than the He SSM).

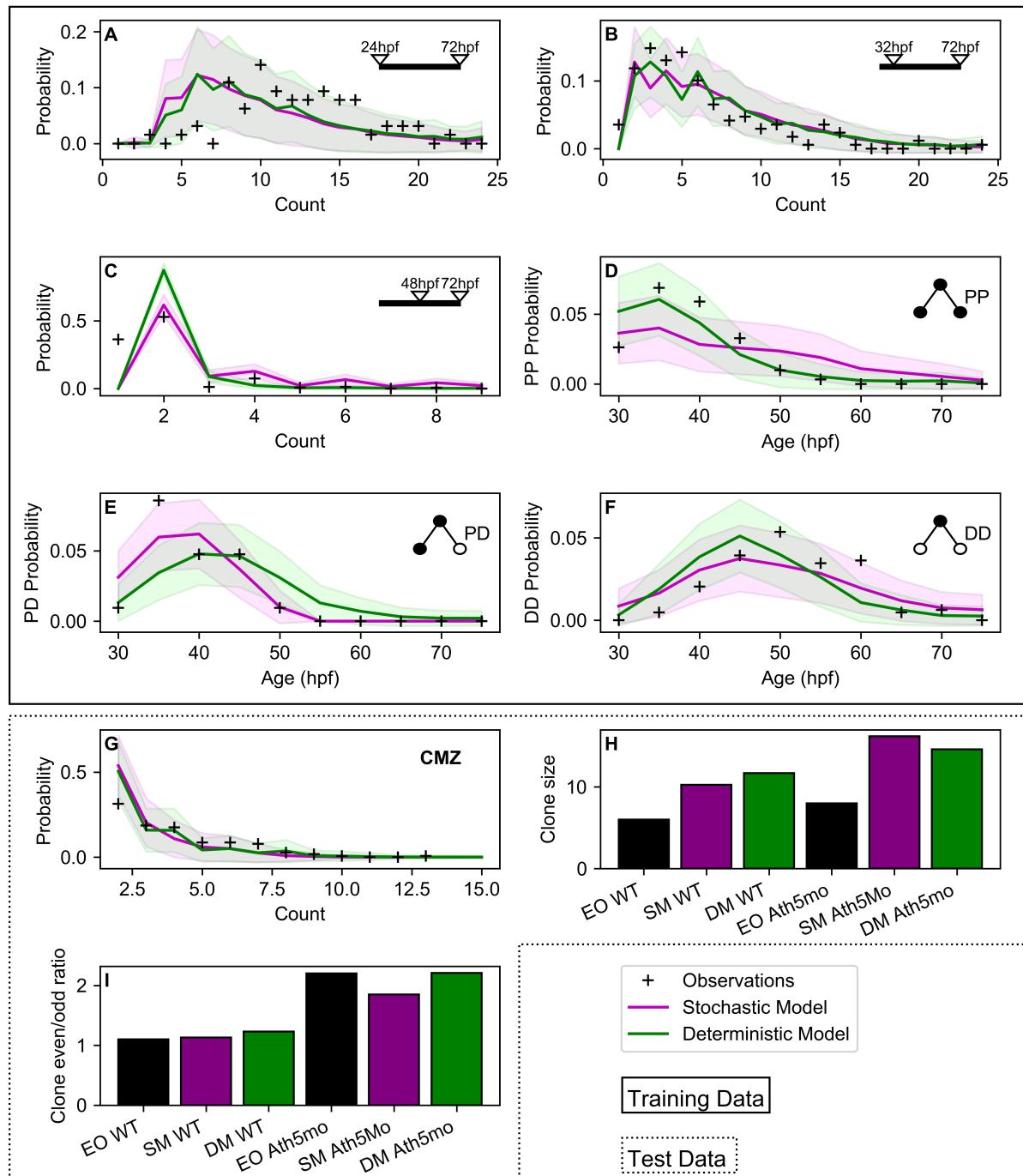


Figure 2.4: Model comparison: the SPSA-optimised He SSM and a deterministic alternative

Empirical observations (black crosses and bars) and SPSA-optimised model output (magenta, stochastic mitotic mode model; green, deterministic mitotic mode model). Model output in panels A-G is displayed as mean \pm 95% CI. Panels A-F: Training dataset, to which models were fit. Panels A-C: Probability of observing lineages of a particular size ("count") after inducing single RPC lineage founders at (A) 24, (B) 32, and (C) 48 hours with an indelible genetic marker, tallying their size at 72hpf. Panels D-F: Probability density of mitotic events of modes (D) PP, (E) PD, and (F) DD over the period of retinal development observed by He et al. Panel G: Probability of observing lineages of a particular size ("count") originating from the CMZ; RPCs are taken to be resident in the CMZ for 17 hours before being forced to differentiate, lineage founders are assumed to be evenly distributed in age across this 17 hour time period. Panel H: Average clonal lineage size of wild type and Ath5 morpholino-treated RPCs. Ath5mo treatment is taken to convert 80% of PD divisions, which occur in the second phase of the He model, to PP divisions, resulting in larger lineages. Panel I: Ratio of even to odd sized clones in wild type and Ath5 morpholino-treated RPCs.

After (re)-fitting to the training dataset, we calculated AIC for the He SSM (hereafter SM, for stochastic model) and our deterministic mitotic mode alternative (hereafter DM), for both training and test datasets. The combined results are displayed in Fig 2.4, with the output of the two models being presented separately in S2 Fig and S3 Fig. Remarkably, the DM closely recapitulates the output of the SM for both datasets. Moreover, while the more highly-parameterised SM permits a better fit to the training data, the DM proves to be a better fit to the test dataset, as summarised in Table 2.1. This is, therefore, a classic case of model overfitting: a higher-parameter model fits some training dataset better than a simpler model, but fails upon challenge with a new dataset. Given this model selection scheme, it is plain that we should choose the DM over the SM as a superior explanatory model, both on the basis of explanatory power and of Occam’s Razor.

Table 2.1: **AIC values for models assessed against training and test datasets**

Model	Training AIC	Test AIC
Stochastic (He fit)	-93.26	255.64
Stochastic (SPSA fit)	-93.80	92.24
Deterministic (SPSA fit)	-69.33	86.60

AIC: Akaike’s information criterion. Lower values reflect better model explanations of the noted dataset. Lowest values are noted in bold.

We therefore conclude that, had the Harris group employed standard model selection procedures, comparing plausible alternatives to their favoured explanation, they would have been forced to conclude that the SMME is not the best available explanation for variability in zebrafish RPC lineage outcomes. It is clear from this analysis that the assumed, but unexplained, linear succession of mitotic mode phases is what provides the overall structure of the model output. Variability in lineage outcomes may be supplied by entirely different model ingredients without any loss of explanatory power (indeed, with some improvement, in our case). The rhetorical emphasis on a stochastic process governing mitotic mode, ostensibly ruling out other types of explanation, is unjustified. It is likely that any number of different types of SSMs, representing other sorts of processes (such as asymmetric segregation of fate determinants, or differential spatial exposure to extracellular signals), can produce identical model output. Given this, we conclude that the SSM model type is simply inadequate for the task of locating the source of variability in RPC lineage outcomes.

2.2.5 SMME SSMs cannot explain the post-embryonic phase of CMZ-driven zebrafish retinal formation

The zebrafish retina, like other fish retinas, and unlike the mammalian retina, continues to grow long after the early developmental period, indeed, well past the organism’s sexual maturity. This may be unsurprising, given that zebrafish increase in length almost ten-fold over the first year of life[PEM⁺⁰⁹], necessitating a continuously growing retina during this period. In fact, the quantitative majority of zebrafish retinal growth occurs post-metamorphosis, outside of the early developmental period. This growth occurs due to the persistence of a population of proliferative RPCs present in an annulus at the periphery of the retina, called the ciliary or circumferential marginal zone (CMZ), which plate out the retina in annular cohorts. A typical “tree-ring” analysis from our studies, marking the DNA of cohorts of cells contributed to the retina at particular times with indelible thymidine analogues, is

shown in Fig 2.5. It is immediately obvious from such experiments that the structure of the zebrafish retina is quantitatively dominated by contributions from the CMZ in the period between one and three months of age. Why peripheral RPCs in zebrafish remain proliferative, while those in mammals are quiescent[Tro00], and whether and how their behaviour might differ from embryonic RPCs, remain unresolved. Answers to these questions may have significant fundamental and therapeutic implications, especially given e.g. the possibility of harnessing endogenous, quiescent, peripheral RPCs in humans for regenerative retinal medicine.

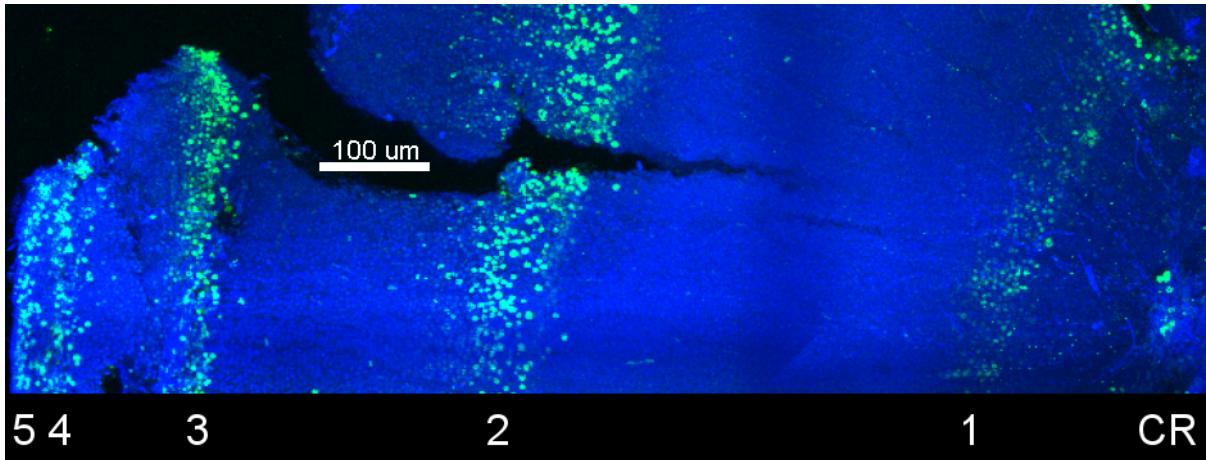


Figure 2.5: Most zebrafish retinal neurons are contributed by the CMZ between one and three months of age

Maximum intensity projection derived from confocal micrographs of a zebrafish whole retina dissected at 5 months post fertilisation (mpf). The animal was treated with BrdU at 1, 2, 3, 4, and 5 mpf for 24hr. Anti-BrdU staining of the whole retina reveals the extent to which the CMZ (which is responsible for retinal growth after approximately 72hpf) contributes new neurons in tree-ring fashion, extending out from the center of the retina (CR), which is formed before 72hpf. Monthly cohorts are labelled appropriately. Scale bar, 100 μ m.

Indeed, the SMME SSMs were originally brought to our attention when the He SSM was deployed in Wan et al.[WAR⁺16], with the claim that the He SSM explains the behaviour of CMZ RPCs. Wan et al. argue that a slowly mitosing population of bona fide stem cells, at the utmost retinal periphery, divides asymmetrically to populate the CMZ with He-SSM-governed RPCs. In other words, the usual suggestion that CMZ RPCs undergo a somewhat different process than embryonic RPCs, perhaps recapitulating across the peripheral-central axis some progression of states or lineage phases that embryonic RPCs pass through in time[HP98], is repudiated in favour of one model which describes the behaviour of all RPCs throughout the life of the organism, with the addition of a small population of stem cells to keep the CMZ stocked with RPCs. If so, this might suggest that the problem of activating quiescent stem cells in the retina is simply that- one need only sort out how to throw the proliferative switch in these cells, since the proliferative and fate specification behaviours of the resultant RPCs will reliably be the same as those observed in development.

While we determined that the SMME is not the best available explanation for RPC lineage outcomes, we still felt that the SSMs associated with this explanation might be used to elucidate this point. In particular, if it is the case that the He SSM provides good estimates of lineage size and proliferative

dynamics in the early zebrafish retina, it should be possible, using this model, and the estimates of putative stem cell proliferative behaviour provided by Wan et al., to simulate the population dynamics of the CMZ, at least through the first few weeks of the organism's life.

In pursuing this point, we noted a peculiar feature of the He SSM not documented by any of the SMME reports: the cell cycle model overstates the *per-lineage* rate of mitoses by as much as a factor of 3. That is, the mitotic mode rate data presented in He et al., recapitulated here in Fig 2.4, panels D, E, and F, and used to optimise both the SM and DM, are probability density functions that are not standardised on a per-lineage basis. These data simply indicate the distribution of mitotic events of a particular type. When we take all of the mitotic events documented by He et al. and calculate the probability of any such event occurring per lineage, per hour, we obtain the values presented in Fig 2.6.

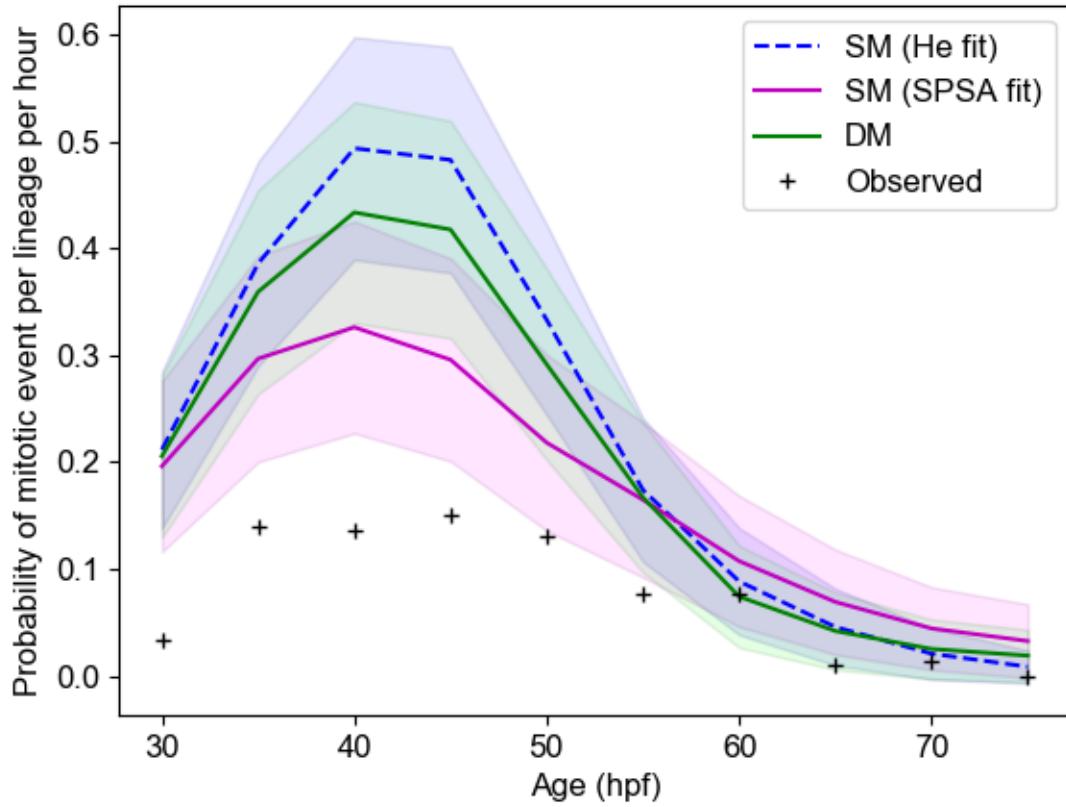


Figure 2.6: **Per-lineage probabilities of mitoses, He et al. observations compared to model output**

Probability of observing a mitotic event over the period studied by He et al., per hour, per lineage. Model output is presented as mean \pm 95% CI.

Similarly, when we performed cumulative thymidine analogue labelling of the 3dpf CMZ, (using the assumptions of Nowakowski et al. [NLM89], which treat the proliferating population as a homogenous, and dividing asymmetrically; these assumptions are flawed but adequate for a rough estimate) we obtain an average cell cycle length of approximately 15 hours, more than twice as long as the He SSM's mean

cycle length. These data are displayed in 10.2. Therefore, the He SSM (in both its original and refit parameterisation, as well as the deterministic alternative, since they all rely on the same proliferative model elements) substantially overstates the proliferative potential of both embryonic and early CMZ RPCs. Still, because we observed a massive build-up of proliferating RPCs between two the first few weeks of post-embryonic CMZ activity involve a massive build-up of proliferating RPCs between two and four weeks post-fertilisation, we thought this might actually suit this later context better.

In order to produce estimates of total annular CMZ population in zebrafish retinas over time, we counted proliferating RPCs present in central coronal sections of zebrafish retinas throughout the first year of life, treating these as samples of the annulus, and calculated the total number of cells that would be present given the diameter of the spherical lens measured at these times. Our simulated CMZ populations were constructed at 3dpf by drawing an initial population of RPCs governed by the He SSM (using the original fit parameters, which further exaggerate the proliferative potential of these lineages) from the observed distribution. An additional number of immortal stem cells amounting to one tenth of this total was added. This is likely an overestimate, given that these putative stem cells are thought to be those in the very peripheral ring of cells around the lens, of which typically two to four may be observed in our central sections with an average of over one hundred proliferating RPCs. Moreover, these simulated stem cells were given a mean cycle time of 30 hours, proliferating about twice as quickly as Wan et al. suggest. Finally, to reflect the fact that these stem cells contribute to the retina in linear cohorts[CAH⁺14], and more of them are therefore required as the retina grows, the stem cells were permitted to divide symmetrically when necessary to maintain the same density of stem cells around the annulus of the lens. Two hundred such CMZ populations were simulated across one year of retinal growth.

The results of these simulations are displayed in Fig 2.7, overlaid over CMZ population estimates derived from observations. Given the generous parameters of the population model, consistently overestimating the proliferative potential of embryonic and early CMZ RPCs, it is surprising that the He SSM proves completely unable to keep up with the growth of the CMZ in the first two months of life. Indeed, the unrealistically active stem cell population the simulated CMZs are provided with is unable to prevent a near-term collapse in RPC numbers, only catching up after the months later as the number of stem cells increases with the growth of the lens. Thus, even given permissive model parameters, the He SSM is not able to recapitulate the quantitatively most important period of retinal growth in the zebrafish.

This analysis strongly suggests that observations of RPCs in embryogenesis and early larval development are unlikely to provide a good quantitative model of the development of the zebrafish retina, even abstracting away spatial and extracellular factors as SSMs necessarily do. Our data point to a second, quantitatively more important phase of retinal development, between approximately one and four months of age, in which RPCs are far more proliferatively active than in early development. It is likely that models that closely associate proliferative behaviour with fate specification, like those of the SMME, will necessarily be unable to explain this period. Recent evidence suggests that mitotic and fate specification behaviours in RPCs may be substantially uncoupled[ESY⁺17]. This would permit the CMZ population to scale appropriately with the growing retina. Alternatively, it is also possible that RPCs are substantially heterogenous with respect to their proliferative behaviour, and that this heterogeneity is mainly apparent later in development.

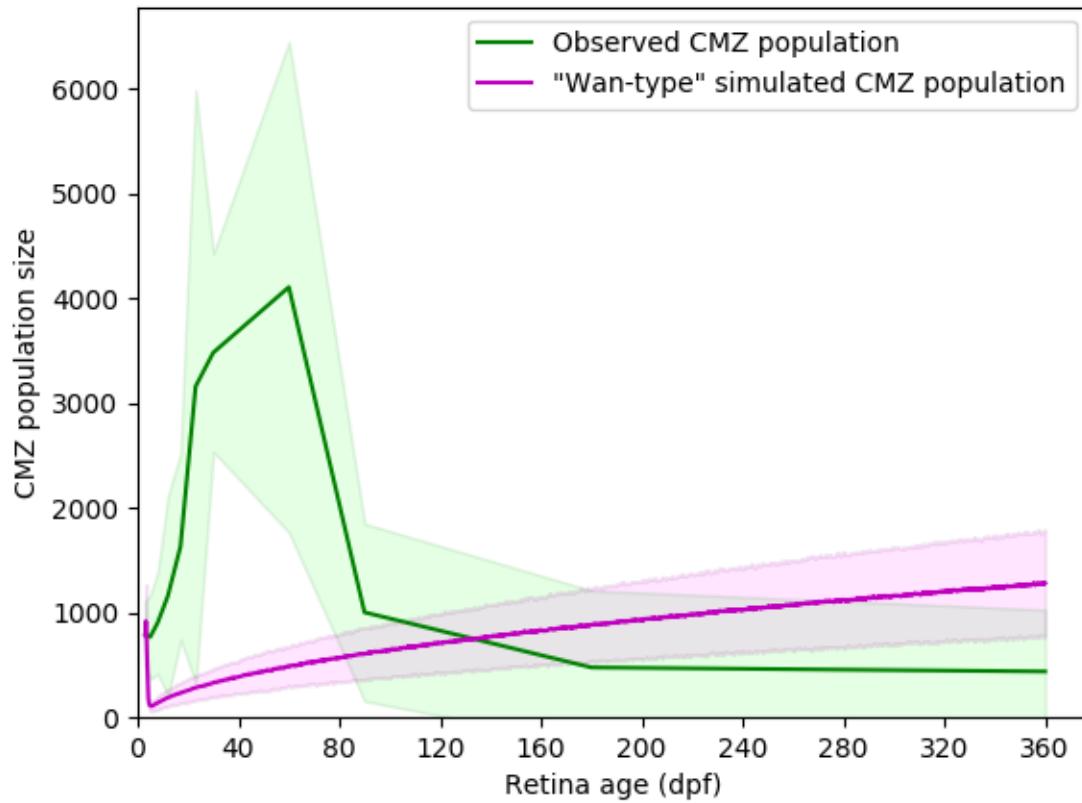


Figure 2.7: CMZ population of proliferating RPCs: estimates from observations and simulated Wan-type CMZs

Annular CMZ population was estimated from empirical observations of central coronal cryosections stained with anti-PCNA, a marker of proliferating cells, standardising by the diameter of the spherical lens observed in these animals. "Wan-type" CMZs were initialised with a number of RPCs drawn from the observed 3dpf population distribution, and given an additional 1/10th of this number of immortal, asymmetrically dividing stem cells, as described in the text, and simulated for the first year of life. All data is presented as mean \pm 95% CI.

2.3 Conclusion

Simple stochastic models are familiar tools for stem cell biologists. Introduced by Till, McCulloch, and Siminovitch in 1964, they proved their utility in describing variability in clonal lineage outcomes of putative stem cells, originating from macromolecular processes beyond the scope of cellular models (and beyond the reach of the molecular techniques of the time). More prosaically, their simplicity afforded computational tractability in an era when processing time was relatively scarce, allowing early access to Monte Carlo simulation techniques. That said, their abstract nature necessarily emphasizes an aspatial, lineage-centric view of tissue development, which cannot account for the generation of structurally complex tissues like retinas beyond cell numbers, and, perhaps, fate composition. As a result of this relatively loose relationship to the complex morphogenetic environment, there are few

constraints on the model configurations that may produce similar outputs. As we hope has been made plain by the analyses herein, this can result in modellers being led astray by their apparently good fits to observations. This is one reason why it has long been unacceptable in other biological modelling communities (particularly, among ecologists) to present the fit of one highly parameterised model as evidence for some theory; it is very rare that there is not a model representing an alternative theory that cannot be made to produce a reasonable model fit. Indeed, as we demonstrate here, the use of standard model selection techniques may make plain that a completely contradictory theory is a better explanation for the data.

To some extent, this can be ameliorated by careful attention to the particular macromolecular or cellular referents that particular model constructs represent. The “mitotic mode” construct present in all SSMs is a particularly ambiguous and problematic one in this regard. “Mitotic mode” is not, itself, a property of a mitotic event, but is rather a retrospective classification of the event after an experimenter observes whether progeny resulting from the event continue to proliferate. Its original appearance in SSMs was simply to allow calculation of clonal population sizes; it was never intended to represent a particular type of process or “decision” made by cells at the time of mitosis to continue proliferating or not. Any number of pre- or post-mitotic signals and processes may result in a particular mitosis being classified as PP, PD, or DD, without anything about the mitotic event itself determining this. The use of such a retrospective classification, rather than the identification of some physical property of the mitotic event (such as the asymmetric inheritance of fate determinants), is straightforwardly a concession that the actual macromolecular determinants of cellular fate are outside the scope of the model.

The SMME represents an attempt to connect this abstract, retrospective model construct to observations of noisy gene transcription[Rv08]. Motivated by the observation that momentary mRNA transcript expression in RPCs is highly variable from cell-to-cell[TSC08], the suggestion is that this transcriptional variability may be responsible for the observed variability in RPC lineage outcomes. Since the extent of transcription noise may be “tuned” to a degree by e.g. promoter sequences[RO04], parameters related to this noise could plausibly be under selective pressure. There are two significant problems with identifying the SMME’s stochastic mitotic mode with this type of process. The first we have demonstrated by constructing an alternative model with deterministic mitotic mode but variable phase lengths: the source of variability may be located elsewhere without compromising the explanatory power of the model. It is therefore impossible to determine what sort of process might give rise to variability in RPC lineage outcomes by using SSMs in this fashion. The second, more fundamental problem is that a causal explanation of the presence of the signal, for which noise is a property, is elided entirely in favour of emphasis on “stochasticity”. This is most apparent in the Boije SSM. In this model, Atoh7 and Ptfla TFs are available to provide their noisy signal in the 4th and 5th lineage generations, and at no other time. We do not dispute that a noisy signal may contribute to variability in that signals’ effects; rather, we suggest that it is the temporal structure of such a signal (assuming this structure can be empirically demonstrated, rather than assumed) that calls for causal explanation. The SMME reports explicitly disclaim the necessity for “causative hypotheses” in the case that a stochastic model provides a good fit to observations. As Jaynes remarked in his classic text on probability theory, “[stochasticity] is always presented in verbiage that implies one is describing an objectively true property of a real physical process. To one who believes such a thing literally, there could be no motivation to investigate the causes more deeply ... and so the real processes at work might never be discovered.”[JBE03]

In identifying the model construct with the physical processes determining RPC outcomes, the SMME

obscures what seems to us to be the primary lesson to be drawn from the Harris groups' beautiful *in vivo* studies of zebrafish RPCs. That is, compared to late rat RPCs in dissociated clonal culture, RPCs in intact zebrafish retinas produce far more orderly outcomes. To the extent that these outcomes are variable, the source of variability remains unidentified. We suggest that, in order to produce good explanations for both the order and unpredictable variability exhibited in RPC behaviours, more sophisticated models and more rigorous modelling practices are required. Resort to "stochasticity" as an explanatory element should not be made in the absence of model comparisons that rule out alternatives with well-defined causal structures, lest we fall into the trap Jaynes warned us about.

Chapter 3

Toward a computational CMZ model comparison framework

3.1 SMME Postmortem: a wrong turn at Gomes

Let us return to the question posed in [Section 1.6](#): has Harris succeeded in finding order by blurring out the chaotic welter of mechanistic explanations for RPC function in retinogenesis? [Chapter 2](#) argues that it has compounded several modelling failures to obscure the basic reality of the matter: the SMME models do not support an explanation based on the noise of some macromolecular process, they rather support explanations based on the original idea of a linear, deterministic series of stages through which RPCs progress, originally put forward by Cepko et al. [[CAY⁺96](#)]. This is the fundamental model ingredient that produces the structure resembling the data, not the various random variables associated with mitotic mode, which we have proven by demonstrating that a deterministic progression of mitotic mode models the data better than its stochastic counterpart. Only by the process of counterinduction, the comparison of models with different propositional structures about reality, does this become clear.

The critical failing of the SMME models is where they depart from the original Gomes SSM: the assumption of the linear structure of temporal phases. This has to be done in order to add the early contribution of RGCs in the modelled zebrafish neurons, and is an essential explanandum for any explanation purporting to relate to retinogenesis; to the extent that it is left without some biological rationale, retinogenesis remains unexplained. It follows that the SMME does not achieve its lofty aim of being a complete description of the aspects of retinogenesis an [SSM](#)[SSM] is capable of explaining, and this is confirmed by our observation that the Wan model [[WAR⁺16](#)] has no explanatory power outside the first few days of life, which makes up a tiny portion of the total retinal contribution to the zebrafish retina.

We must also assess that Harris' first theoretical maneuver, explaining the data with a model whose structure supports a stochastic resolution of mitotic mode "decisions", fails in the face of an alternative which locates stochasticity elsewhere. Simply by introducing variability into the lengths of the linear temporal program already assumed by the SMME models, we can remove variability from the mitotic mode and produce a superior model. The underlying data used to inform these models speak better to any entirely different selection from the array of theoretical options outlined in [Section 1.2](#), the linear progression of competencies. Because the SSM model form does not usually have spatial dimensions, we did not test models involving variability in extracellular signals, but it is a good bet that such a model

could be made to fit about as well as either of the ones tested in the previous chapter. In effect, if we are to follow Harris' inferential logic, variability in RPC outcomes can be explained by variability in virtually any parameter which affects fate outcomes. The explanation collapses into itself, which is why Boije et al. is forced to defend their idiosyncratic definition of "stochasticity" at length: variability is being used to explain variability. If "randomness" or "stochasticity" is accepted as a property of existents, rather than representations of our uncertainty about them, all biological variability is trivially explained by referring to it. Because of the seriousness of this problem for biological inferences¹, an explanation of the Bayesian epistemological view of probability has been provided in ???. Moreover, thorough explanation of the sense in which there is no good argument for "randomness" being a property of real existents is provided in ???. It suffices here to conclude that variability in RPC outcomes is not well explained by the SMME models, in part because their interpretation depends entirely on an incorrect understanding of the concept of stochasticity, and in part because they were never tested against alternatives.

The second theoretical maneuver was to nominate a particular macromolecular system as the physical locus of the stochastic process, in this case represented by *Atoh7* and *Ptf1a* as labels for abstract Bernoulli distributed processes. It remains obscure in what sense the model variables relate to their namesake transcription factors (transcription of the TF itself? activation of other genes?). Plainly, these factors are involved in relevant RPC behaviours, but it is unclear why they have been nominated as causally upstream of the "mitotic mode" selection. Since the Boije model inherits the assumption of a linear progression of stages, it is very likely that a model with similar explanatory power could be built using the same strategy outlined above, locating random variability outside mitotic mode, although this would be superfluous.

On the basis of the above, we can conclude that the sole formally testable model of RPC function in the zebrafish retina is not adequate for our purposes. But how did we get here? As noted in Section 1.3.2, the notion that the most significant proliferative and specificative phenomena associated with RPCs are produced by mechanisms intrinsic to the cells derives much of its empirical support from the Raff group's work. This story began developing with the observation that co-culturing E15 rat RPCs in dissociated pellet cultures with P1 cells did not accelerate the appearance of the first rods derived from the E15 progenitors (which occurred at a similar time as *in vivo*), suggesting that the specificative "schedule" of these cells is at least partially intrinsic² [WR90]. The scope of these observations were dramatically expanded by Raff's subsequent work, intended to address the relative significance of intrinsic versus extrinsic processes in RPC function by comparing clonal RPC lineages in fully dissociated clonal-density cell culture to those in intact explants [CBR03]. The remarkable finding of this study was that dissociated E16-17 rat RPC lineages produce very similar numbers and types of retinal neurons, albeit without morphological or molecular markers of mature neurons. This observation provided strong evidence for the predominant importance of RPC-intrinsic processes in determining both the proliferative and specificative outcomes for late RPC lineages, since the complex, spatially organised context of intact explanted tissue seemed to only be required for the maturation of neurons, and was not required to regulate proliferation or the initial commitment to an appropriate distribution of lineage outcomes. As these are the two most obvious and critical parameters that must be achieved for RPCs to produce a functional retina of the appropriate size, the suggestion that both

¹Having experienced well-respected stem cell biologists confidently asserting that Harris' work proves that RPC mitotic outcomes are 'spontaneous', ie. causeless and without explanation, I believe these studies have been both influential and confusing for many.

²Co-culturing with P1 cells, did, however, significantly increase the proportion of E15 RPCs specified as rods, resulting in Raff's suggestion here that both intrinsic and extrinsic factors are important.

are largely determined by intrinsic processes seemed a striking confirmation of Williams and Goldwitz's much earlier suggestion [WG92], against the prevailing view of the day, that lineage had a greater role to play than cellular microenvironment in RPC contributions. Interestingly, Raff's interpretation of their 2003 data was that RPCs were most likely stepping through a linear, programmed developmental sequence rather than undergoing shifts in the probabilities of variable outcomes over time. It is notable that this interpretation arises not from the data collected in their study, but from considerations of a single unusual clone reported by [TSC90], and by analogy with drosophila neuroblasts. In retrospect, these arguments for linear sequences of deterministic RPC outcomes do not seem particularly strong, and it is perhaps unsurprising that these studies are remembered mainly for highlighting the importance of RPC-intrinsic processes.

The Gomes study, with its detailed study of particular rat late-embryonic RPC lineages, thus seemed to solidify the notion that unpredictable RPC-intrinsic processes dominate lineage outcomes [GZC⁺11]. But this study, like its forebears originating in Raff's work, is concerned with a population of RPCs that is too old to produce RGCs. This is the essential point: the SMME does not even try to explain the early appearance of RGCs in terms of some macromolecular mechanism, although this is the single most significant difference between the zebrafish RPC lineages studied by Harris and the rat lineages studied by Raff and Cayouette [CBR03, GZC⁺11]. It is only by assuming the unexplained linear temporal structure of RPC specification that Harris is able to continue the rhetorical focus on the stochastic elements of the model, and so is lead down the garden path of spurious model fits and logically unsound interpretation of model structure.

3.2 Implications of the SMME's failure for modelling CMZ RPCs

Having taken seriously the possibility that an adequate model of zebrafish retinogenesis already exists, and concluded in the negative, we must proceed to a plan to rectify this state of affairs.

Firstly, the obvious lessons in statistical acumen must be learned. The statistical practices used in the SMME reports are not acceptable by any contemporary standard, and would not have been accepted in another field where practitioners are more familiar with modern statistical techniques. We must be dissuaded of the notion that a single, hand fitted model has any explanatory power at all; it may be an exercise in modelling prowess, but it is not a legitimate part of the quantitative scientific discourse unless some statistical property of the model is actually computed³. Moreover, to have any confidence about our interpretation of the model, we must test alternatives that make fundamentally different assumptions about the causal structure of the phenomenon being explained by the models.

Can we conclude that the approach used in Chapter 2 is adequate to our task? There are two broad elements to consider here: the appropriateness of the overall modelling approach represented by the SSM in relation to the hypotheses we wish to test, as well as the soundness of the statistical procedures.

Taking up the SSM, by far its most attractive features are its computational efficiency and the ease of producing a custom model to fit some particular case- for all their failings, the SMME models include some elements like the correlation of cell cycle lengths in mitotic sisters that greatly improve the temporal modelling of RPC lineage outcomes, for instance.

The observations arising from the SMME strongly suggest that we would like to test hypotheses

³The commonly maligned Fischerian t-test procedure [HKB08, pp.181] is, in this sense, better than simply plotting some model output over observations.

about RGC specification in order to find better models of RPC function. Carefully reconsidering the hypothesis of Neumann et al. [Neu00], that Shh from nearby RGCs induces cell cycle exit and RGC specification, would seem to be a high priority, for instance. Can such a scenario be represented in an SSM? One could model the probability of being within Shh induction range of an RGC in the early part of a lineage's life, for instance. Doubtless, a model that incorporated variable RPC specification outcomes determined on this basis could be made to fit data about as well as the variable mitotic mode or variable phase length models tested above. That said, it is not very clear that the aspatial data to which SSMs are fitted is capable of constraining the likelihood of model alternatives that include spatial components. While we determined that our deterministic mitotic mode model fit test data somewhat better than the stochastic model, the modest size of the calculated AIC differences suggests that comparing SSMs is not a particularly good way to distinguish even alternative hypotheses about the structure of processes the SSM models explicitly, like cell cycle length and mitotic mode.

We can therefore predict that we will be unable to test important models, involving documented macromolecular explanations of RGC specification, without the ability to represent some spatial information. Still, the computational benefits of the SSM are too appealing to leave out of the toolkit entirely. Taken together, this suggests that we need a very general method of testing models of potentially very different structures against one another.

Turning then to our statistical procedures, are these adequate to our goals? We can broadly characterize the approach taken as estimating the local optimum of a loss function for model output, given the dataset; specifically, minimizing Akaike's information criterion by simultaneous perturbation stochastic approximation (SPSA). This procedure has some advantages. AIC is a well-understood measure, well-grounded in information theory, which penalizes superfluous model complexity in a consistent and rigorous way. SPSA is a widely used, well understood algorithm that can be applied to any reasonable euclidean parameter spaces; cases where parameter spaces are bounded (typical of biological models where negative parameter values are usually nonsensical) are explicitly accounted for, and so on. This procedure is better than many that are available.

Still, after the experience of the SMME model comparison, a certain uneasiness is warranted. The AIC calculation is, after all, based on a single estimate for the locally optimal parameterisation of the model. Relative AIC rankings, then, are strongly influenced by the "well depth" of the AIC surface at the local minimum in parameter space. We can easily imagine a case where a model with a very narrow range of parameter space that fits data very well appears to be better, by the AIC metric, than one that fits the data slightly more loosely, but over a much broader range of the parameter space. The first model is a "fragile" fit, probably depending heavily on the particular structure of the training dataset, while the second would likely be much more robust across a range of observations; still, in this case, AIC would lead us to select the fragile model over its robust counterpart. Indeed, blind interpretation of AIC rankings has lead to its use in ecology being described as a "cult" [BB20]. It is easy to imagine practitioners being reduced to "AIC hacking" in the same manner that "p hacking" occurs in order to achieve some arbitrary value for a hypothesis.

Moreover, while SPSA is an eminently practical algorithm, useful in a wide variety of contexts, its statistical guarantee is only that it will almost-surely find the local minimum of the loss function. While this will be good enough in some applications, for highly parameterised models with complex loss function surfaces, there will be many local minima for SPSA to get "stuck" in that are nowhere near the

global optimum⁴. If we are evaluating hypotheses in order to make decisions about potentially years-long research projects, more certainty about the reliability of the method is necessary. Finally, while SPSA is requires much less computational effort than more global Monte Carlo parameter estimation techniques like simulated annealing or Hamiltonian Monte Carlo, it requires about as much application-specific tuning.

Fortunately, a complete system of Bayesian inference which addresses all of the problems mentioned above has been promulgated over the last 15 years: nested sampling. Originally introduced by John Skilling [Ski06], this use of this system has become widespread in cosmology [Tro08], where its generality and ability to cope with complex, high dimensional parameter spaces has been well proven.

3.3 Desiridata for spatial CMZ models in a putative model comparison framework

The simulations presented in ?? consisted solely of abstract collections of lineages. The members of these model colonies have no activities beyond proliferation and no functional attributes beyond their proliferative status⁵. Despite the underlying code consisting of relatively high performance C++, these simple models nonetheless occupied the local component of the cluster used in this work for some weeks. We can therefore see how a simple approach to ranking basic models against a smallish dataset approaches the reasonable limits of what most labs are likely to be able to achieve with any given machine in a month or so. Because the introduction of spatial simulation implies, perhaps, an order of magnitude more computational time, we may plausibly be limited to one inference based on model comparison per year with local resources. The problem of computational limits to model selection is discussed in the relevant section of the Technical Appendix, so it will suffice here to state that this constraint dominates all other considerations in the selection of the overall boundaries within which we intend to build models of the CMZ. In any environment where funding constraints limit the cloud-export of computational burden from the confines of the research institution, it is reasonable to proceed upwards in model computational expense from the cheaper-but-inadequate in search of the adequate-but-still-affordable. In this case, we move from the aspatial SSM into the realm of explicit spatial modelling, seeking the minimum model complexity required to compare hypotheses of interest to us.

3.3.1 Spatial dimension of the models: the "slice model"

Although decomposing the RPC population of the CMZ into a collection of unordered, independently-proliferating SSMs prevents us from assessing many interesting hypotheses, a computational model of the entire CMZ or retina may not be required to compare many interesting hypotheses. Because numerous observations suggest that individual RPC lineages contribute to the retina in linear cohorts of neurons, it follows that any particular centrally-oriented slice of the CMZ annulus will be responsible for the generation of the neurons central to it. Conceptually, then, the retina can be thought of as a series of these "slice units", lined up radially like slices of pie. A complete "slice unit" would include the central-most larval remnant, contributed by embryonic retinogenesis, surrounded by the CMZ's more ordered neural contribution from the postembryonic period, and, peripherally, the CMZ itself. Depending on

⁴This is part of what is meant by "the curse of dimensionality", when speaking of the difficulty of sampling the loss function in high dimensional parameter spaces.

⁵I.e. the specified identity of post-proliferative cells in these models has no function within the model.

the hypotheses to be tested, the differentiated central retina may be mostly irrelevant, so the modelled slice may consist mainly of the CMZ and its interface with these central neurons. It is important to note that these slices are conceptually different from the linear cohorts generated from particular lineages, the so-called ‘ArCCoS’, which justify the slices. It is not necessarily the case that a slice model would consider only one RPC lineage; the slices are better thought of as spatial boxes that sample the CMZ and adjacent central neurons, the conceptual equivalent of the histological section through the eye⁶.

Slice models are especially attractive because the observed proliferative status, position, specified fate, etc. of simulated cells can be easily related to the summary statistics used to describe fixed sections of retinal tissue in this thesis and elsewhere. If our histological sections are of an appropriate width, the slice model can be selected to produce output that is directly comparable to observed histological results. This is especially important for studying the postembryonic CMZ, which rapidly becomes optically inaccessible due to the increasing thickness and pigmentation of overlying tissue, so that live imaging cannot be used ⁷.

If the retina can be usefully thought of as a series of slice models, and the activity of the CMZ is basically homogenous around its circumference, it follows that the activity of the CMZ can be usefully represented with a single such model. Moreover, the slice need only include one portion of the retinal periphery, the orientation of which is irrelevant (i.e. the model parameterisation for the dorsal portion of a coronal slice is identical to the ventral). It may be erroneous to abstract such a slice from its context, because the modelled cells are in contact with adjacent slices. However, if the CMZ homogeneity premise is valid, this can easily be incorporated into the model by providing a layer of “ghost” cells on either side of slice whose parameters are determined by the slice itself ⁸. That is, given the premises, a slice model which interacts with copies of itself is a complete model of CMZ-driven retinogenesis.

All this said, the zebrafish retina is a manifestly asymmetrical structure, with an optic nerve positioned ventro-temporally relative to the center of the optic cup. The CMZ itself is generally understood to be an asymmetric structure, with a larger dorsal than ventral population. This calls into question the assumption of CMZ homogeneity outlined above. It is nevertheless plausible that the appearance and maintenance of these structural features of the zebrafish retina could be explained with a set of slice models, for example, one for each of the dorsal, ventral, nasal, and temporal extrema. Ultimately, these considerations only relevant if our objective is to compare model explanations for retinogenesis as a whole. If we restrict ourselves to the more modest goal of, say, ranking causal influences on the proliferative and specificative behaviours of RPCs in the dorsal extremity of the CMZ, this could provide significant insight into the regulation of peripheral stem cells in other species. In this sense, a single slice model could still be valuable even if it fails to explain aspects of tissue-level zebrafish retinogenesis.

3.3.2 Temporal resolution of the models

Another important consideration for any CMZ model comparison framework is the time-scale of any phenomena which are to be admitted as possible causal contributors to retinogenesis. While it is reasonable to think that proliferative events may be well-described with a model that operates on a scale of days and fractions thereof, and that such a model would be well suited to describing the full sweep

⁶The case of only one simulated lineage may still arise given thin enough sample boxes or old enough animals, but it is a limiting case and probably would not be the norm.

⁷It is worth noting that the spatial parameters of such models would pertain to fixed and not live retinas; it is possible that some scaling relation compensating for fixative shrinkage could allow the inclusion of live imaging data.

⁸This is implemented in CHASTE as “ghost nodes”.

of CMZ activity across the life of the organism, very few of the relevant macromolecular processes are likely to be well-described with a resolution more coarse than seconds or minutes. The implied difference in the number of calculations required to simulate any given time period is thus several orders of magnitude. A simplifying assumption of the temporal homogeneity of CMZ activity would allow us to abstract long developmental time frames; an explanation that pertains to a few hours can perhaps be extended without recalculation.

If an assumption of temporal homogeneity proves inadequate, the functional structure of the simulation must be structured by this consideration. To illustrate this, consider a case where we wish to incorporate measurements of eye pressure, or membrane tension across the retina, as inputs into the proliferative activity of the CMZ, by way of progenitor cortical tension [Win15]. This would permit assessing whether modelling tissue-mechanical inputs to cellular activities allows us to extract additional information from our observations. If such physical model elements are to be included, the functions which relate physical parametric data to the proliferative behaviour of CMZ progenitors must not operate on a time scale too short to reasonably cover the period of interest. Let us suppose we are mainly interested in explaining the assembly of functional units of the mature, specified retina by the CMZ, from the first division of the presumed distal stem cell responsible for the unit to the determination of the last neuron of its functional column. This process certainly takes days; the cumulative thymidine analogue labelling conducted for this thesis never revealed labelled cells in the structured, determined retinal layers within 24 hours of the end of the analogue pulse. Generally, one must wait at least 3 days before reliably finding most of a labelled cohort in the mature retinal laminae. In this case, we might prefer simplifying assumptions about the relationship of measured retinal surface tension to cortical tension's presumed effects on proliferative activity, over a detailed finite element simulation of cortical tension itself with a resolution of minutes.

3.4 Bayesian decisionmaking for structuring models under uncertainty

From the foregoing discussion, we can see that the extent to which model simplifications are justified, and the types of phenomena that could be explained with these models, depend heavily on considerations that can only be informed by observations. As has been demonstrated in ??, the growth of the CMZ population cannot be explained by models fitted to embryonic RPC activity. However, it remains unclear what sort of alternative structure is justified by observations, given our uncertainty about inferred parameters of the populations being measured.

Bayesian statistical methods provide a convenient way to approach this problem. They allow the direct estimation of hypotheses' credibility given data, expressed as a probability. They also permit as well as direct comparison of the evidence for linear regression models, taking into account uncertainty on the model weights. These calculations have straightforward interpretations which allow us to answer questions about the likelihood of the assumptions discussed above actually obtaining at particular times and places in the CMZ.

All of the measurements presented here were obtained from groups of fish of a particular age. Because the measurements are population counts and tissue dimensions in different individual fish, it is assumed that they are the outcome of many independent causal processes. The central limit theorem, therefore, justifies the assumption that the measurements are normally distributed in the population of fish of any

given age.

Given this normal model of measurement distribution in the cohort, uncertainty on the mean and variance of the model is represented with a normal-gamma distribution over those parameters. We then calculate the marginal posterior distribution of the mean, given our uncertainty about the model parameters. These marginal posterior distributions are subsequently sampled by Monte Carlo in order both to calculate derived quantities with appropriate credible intervals, as well as to empirically estimate the probability of various mean comparison hypotheses. Separately, the linear regression technique often known as "Empirical Bayes" was used to perform evidence-based selection on linear models of CMZ proliferation.

The techniques used here are not complex, but they may be unfamiliar to some readers. In addition to the brief summary above, detailed methods can be found in [Section 11.6](#) of the Supplementary Materials, while more extensive theoretical background is available in [Section 13.2.1](#) of the Theoretical Appendix.

Chapter 4

Bayesian periodization of postembryonic CMZ activity

4.1 Preliminaries: Calculation of Bayesian evidence militates for independent log-Normal modelling of CMZ parameters

To take up the question of how CMZ RPC activity evolves over time, we have collected observations of a variety of CMZ and retinal parameters derived from histological studies of cryosections of zebrafish eyes harvested over the first year of the animal’s life. We wish to extract as much information as possible about the structure and time-evolution of the CMZ population as a whole, in order to know how to best model its constituent RPCs. The initial approach here will be to estimate parameters of the whole-eye CMZ from sample cryosections, and to use these estimates as the dataset which will inform our selection of appropriate models.

While it remains common to assume that population data are normally distributed, and so simply to calculate means and standard deviations as the statistical representation of the underlying population, it has been known for decades [Hea67] that log-normal distributions are usually better models of the outcomes produced by additive processes with small, variable steps (like population sizes or income distributions). We therefore start with the selection of appropriate models for the CMZ- and retina-level population measurements critical to the inferences that follow.

As a first pass at this question, we calculate the likelihood ratio for the hypothesis that the parameter measurements, and the calculated quantities derived from them, are log-normally distributed against the one that they are simply normally distributed. These results are displayed in Table 4.1

These results suggest that the organism-level population distribution of eye-level CMZ population counts, as assayed by PCNA immunostaining, is better modelled log-normally than normally. This is true whether we test the primary per-section count measurements, or the estimated whole-annulus population, calculated as described in Section 10.1.4, even though this quantity is calculated using the normally-distributed lens diameter¹. The most likely log-Normal representations of the CMZ populations are about two orders of magnitude more likely than the Normal alternative. Additionally, although normal models

¹The apparent superiority of the normal model for lens diameter may be due to the paucity of data at later time points; as described in INSERT METHODS AUTOREF, lenses are difficult to retain in this histological context.

Table 4.1: Likelihood ratio comparison between normal and log-normal models of retinal population parameters

Parameter	\mathcal{N} logLH	Log- \mathcal{N} logLH	logLR
Sectional PCNA+ve	-285.611	-283.214	2.397
Lens diameter	-203.102	-203.854	-0.752
CMZ annular pop.(†)	-484.768	-482.733	2.035
RPE length	-368.232	-368.609	-0.377
CR thickness (†)	-240.858	-241.032	-0.174
CR volume (†)	-1091.615	-1091.146	0.469

\mathcal{N} : Normal distribution. logLH: logarithm of $p(D|M)$, the likelihood of the data given the model. logLR: logarithm of the likelihood ratio; positive ratios in favour of the log- \mathcal{N} model. Superior likelihoods are bolded. †: Calculated quantities. Sectional PCNA+ve: population of PCNA-positive CMZ RPCs per $14\mu\text{m}$ cryosection. CMZ annular pop: population of annular CMZ. RPE: retinal pigmented epithelium. CR: cellular retina.

are slightly favoured for describing the population-level distributions of RPE length and cellular retina thickness, the derived cellular retina volume quantity is better modeled by a log-Normal distribution. As the two calculated quantities will be the primary ones used in our inferences about population-level CMZ dynamics, we are most concerned with these measures; at this point, the log-Normal distribution looks to be more informative for both.

We have emphasized the danger of relying overmuch on the parameterisation of single most-likely model fits in Chapter 3, which is what the simple likelihood ratios above represent: the joint likelihood of the maximum a posteriori distribution fitted to the measurements taken from each age cohort. Since the choice of model describing the estimated annular population and retinal volume is critical to the success of later inferences, we used Galilean Monte Carlo-Nested Sampling (GMC-NS) to estimate the Bayesian evidence (the marginal probability of the data over all model parameterisations) for these hypotheses. Although it is very unlikely that GMC-NS will result in conclusions from the likelihood ratio, this simple test serves to prove the function of the `GMC_NS.jl` package, and demonstrate the inferential logic. Evidence estimates and ratios are presented in Table 4.2.

Table 4.2: Evidence favours log-normal models of retinal population parameters

Parameter	\mathcal{N} logZ	Log- \mathcal{N} logZ	logZR	σ significance
CMZ Population	-4953.7 ± 7.7	-1148.3 ± 4.5	3805.4 ± 8.9	428.4
Estimated Retinal Volume	-9539.0 ± 40.0	-2337.5 ± 9.0	7202.0 ± 41.0	176.0

\mathcal{N} : Normal distribution. logZ: logarithm of $p(D)$, the marginal likelihood of the data, or model evidence. logZR: evidence ratio; positive ratios in favour of the log- \mathcal{N} model. Largest evidence values bolded. CR: cellular retina.

Unsurprisingly, full estimation of the Bayesian evidence for the Normal vs. log-Normal hypotheses for our calculated parameters produces the same basic story as the rough calculation of likelihood ratio from the single-fit MAP models. There are approximately 3800 orders of magnitude more evidence for the log-Normal model of interindividual variation in estimated CMZ annulus RPC population over time. This result has greater than 420 standard deviations of significance². However, there are approximately

²The typical standard for a "discovery" in particle physics is $>5\sigma$ [Lyo13]. Assuming Normally distributed error, the

7200 orders of magnitude more evidence for the log-Normal model for the variability in the cellular retinal volume estimate, with 176 standard deviations of significance. Given the likelihood ratios alone, we might have suspected that the log-Normal model was more justified for modelling the population data than for the volume data. In fact, with the full estimation of the evidence, it becomes clear that the converse is true, demonstrating how evidence measurements can supply a fuller picture than maximum likelihood methods. In any case, based on these results, we need not have any compunction about modelling population outcomes for these parameters with log-Normal distributions, as the evidence supplied by our observations plainly supports it.

If between-individual variation in retinal CMZ population and retinal volume are both well-modelled log-normally, the question of their independence immediately arises. It seems plausible that the size of the CMZ population would be roughly proportional to the overall volume of the retina, upon central specification, and establishment of the peripheral CMZ remnant. Moreover, since the growth of retinal volume over the life of the organism is driven primarily by the CMZ³, it seems likely that either the retinal growth rate is well correlated with the size of the CMZ. Since both of these questions bear upon the manner in which we model the CMZ, we performed Bayesian model selection by the so-called Empirical Bayes method for linear regression, which provides for direct estimation of the evidence for models consisting of linear equations of variables [Bis06].

We find that individual CMZ population and retinal volume estimates are better described by uncorrelated models at in all ages, with the exception of 23.0 dpf. It is interesting to note that the evidence in favour of non-correlation is weakest between 17 and 30 dpf. These data are displayed in Table 4.3. Of particular importance are the data for 3dpf embryos, as we intend to seed model retinae with CMZ populations and volumes drawn from these distributions. The data for these animals are plotted in Figure 4.1. The general lack of correlation between CMZ population and retinal volume estimates may be due to the loss of information involved in the estimation calculations; on the other hand, it is more plausible that CMZ population should be associated with the rate of retinal volume growth rather than the volume of the retina itself, which suggests that the rate of retinal contribution from the CMZ is probably highest between 17 and 23 days. Because growth rate data are unavailable for single individuals, we cannot make this inference directly.

From these analyses, we conclude that organismal variability in the CMZ population and retinal volume estimates are best described by independent log-Normal distributions. Because log-Normal distributions are simply transformed Normal Gaussian distributions, we may model our uncertainty about their parameters with Normal-Gamma distributions over the mean and variance of the underlying Normal distribution of the log-Normal population model ("the underlying"). That is, our prior and posterior belief about the relative likelihood of values of the mean of the underlying may be modelled with a Normal distribution, and our beliefs about its variance with a Gamma, such that our joint uncertainty is the product of the two distributions. This is explained in more detail in ???. A useful analytic feature of the Normal-Gamma prior is that the marginal posterior distribution of the mean, assuming an uninformative (ignorance) prior, is a location-scaled T distribution; this is so because the

probability that the models were, in fact, equally good at 3 standard deviations of significance would be about a tenth of a percent (i.e. .001). At 10 standard deviations the figure is 7.62e-24. Assuming the sample is representative, we have nigh certainty about these results and no reason to pursue the matter further.

³One observes occasional proliferative clusters in the central retina throughout the life of the fish; these are typically ascribed to Müller glial repair processes. I am unaware of any estimate as to the relative contribution of these clusters vs. the CMZ. As we shall see, there is probably more turnover in the specified retina than previously believed. As a result, the relative contribution of these central clusters should probably be subject to statistical estimation; they may be more significant than mere lesion-repair sites.

Table 4.3: Evidence favours uncorrelated linear models of CMZ-population and retinal volume over time

Age (dpf)	Uncorrelated logZ	Correlated logZ	logZR
3.0	-87.651	-91.902	4.252
5.0	-90.0	-92.362	2.362
8.0	-90.918	-96.013	5.095
12.0	-91.183	-99.049	7.866
17.0	-94.818	-96.668	1.85
23.0	-103.386	-103.219	-0.167
30.0	-103.511	-104.092	0.581
60.0	-115.025	-118.169	3.144
90.0	-113.533	-122.427	8.894
180.0	-116.778	-124.547	7.769
360.0	-121.016	-128.637	7.621

logZ: logarithm of $p(D)$, the marginal likelihood of the data, or model evidence. logZR: evidence ratio; positive ratios in favour of the uncorrelated model. Largest evidence values bolded.

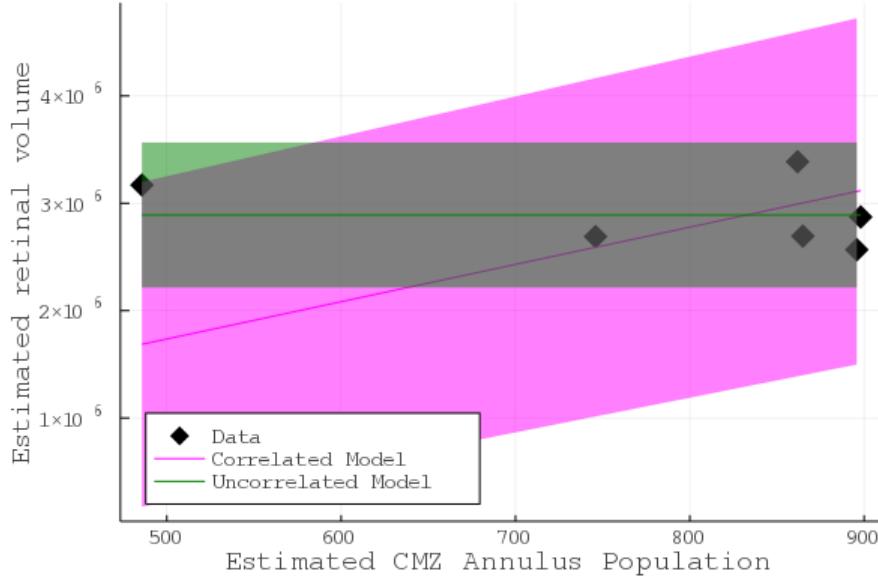


Figure 4.1: CMZ population and retinal volume estimates are uncorrelated at 3dpf
Individual CMZ population estimate vs retinal volume estimate for 3dpf animals. Uncorrelated and correlated linear models of these variables are plotted as the mean \pm 95% CI of the predictive distribution of the fitted model.

weighted sum of an infinite series of Normal distributions (i.e. the likelihood-weighted sum of all the Normal distributions that could underly the log-Normal models), is a T distribution. T distributions are notably more resistant to outlier distortion than Normal distributions themselves are, and may be estimated with as few as two observations, making them highly flexible. Most of the descriptive statistics in the next section, therefore, calculate the credible interval for the posterior mean of the underlying by T distributions (with the correct change of variables by exponential transformation to produce the features of the correct log-Normal distribution). Unfortunately, differences of T distributions are not,

themselves, necessarily T-distributed, so we have relied on Monte Carlo estimation of rates of change of the these posterior means over time. With our log-Normal models selected, and the descriptive tool of the posterior mean T distribution in hand, we turn now to a survey of the zebrafish CMZ in the first year of life.

4.2 Survey of CMZ population and gross retinal contribution

If it is true that the majority of zebrafish retinogenesis occurs postembryonically, and that models trained on embryonic data do not describe this period well, what characterises this CMZ-driven phase of retinogenesis? We begin by presenting our estimates of individual CMZ annulus population and retinal volume over the first year of life, in Figure 4.2, panels A and C.

An initial, relatively quiescent period in the population history of the CMZ can be inferred from the lack of growth observed in the first week of life, as well as the observations in Figure 5.2, which demonstrate that CMZ RPCs are proliferating too slowly to be labelled by a day's pulse of a thymidine analogue at 5dpf in the wild-type and heterozygote siblings of npat mutants. Interestingly, we have good confidence that retinal volume continues to grow over this time; 99.56% of the marginal posterior mass of the mean 5dpf estimate is above the 3dpf mean, suggesting that this proliferative pause is too short to appear in the retinal volume data.

In any case, the first two to three weeks of life (magnified in lens insets in panels A and C) appear to mark a relatively slow build in both estimated CMZ annulus population and retinal volume compared to the explosion which follows immediately thereafter. While zebrafish are better staged by size than age [PEM⁺09], the onset of this explosive growth seems to come somewhat earlier than the typical metamorphic transition from larval to juvenile stages at 45dpf [SH14]. This raises the question of the manner in which the CMZ contributes to the retina during the critical period of exponential growth of the organism, between about 45 and 90dpf. It is plausible, for instance, that the growth of the CMZ population mainly reflects the increased output of stem cells, with RPCs specifying at some steady rate, or that the CMZ builds itself up for a wave of specification somewhat later, similarly to the sequence of events in the embryonic central retina.

In order to get a better sense of this timing, we simulated the mean daily rate of CMZ annulus population and retinal volume change, by performing Monte Carlo difference operations between samples from the marginal posterior means of subsequent timepoints. These simulated mean rates and their associated confidence intervals are plotted in Figure 4.2 panels B and D. These show the basic time-structure of the phenomenon; the large increase in simulated daily CMZ population growth rate occurs before the large increase in retinal volume, but the CMZ population estimate does not begin to drop off until 60dpf, by which time the majority of volumetric growth is complete. This seems to substantiate some combination of the scenarios outlined above: there is an early buildup of CMZ population around 18-30dpf, before CMZ RPCs begin to make their primary contribution to the retina between 30-60dpf, which is characterised by much slower population growth and more rapid volumetric growth, implying steady and elevated specification of RPCs. The greater uncertainty associated with our population estimates, relative to their magnitude, results in correspondingly less certainty about the size of the population growth rate spike. For instance, we calculate a >99% probability that the mean retinal volumetric rate growth peak at 60 dpf is at least 6-fold greater than the mean at 30dpf. By contrast, only about 97% of the posterior mean density of the population growth rate at the 23d peak (230.2 $\frac{\text{cells}}{\text{d}}$)

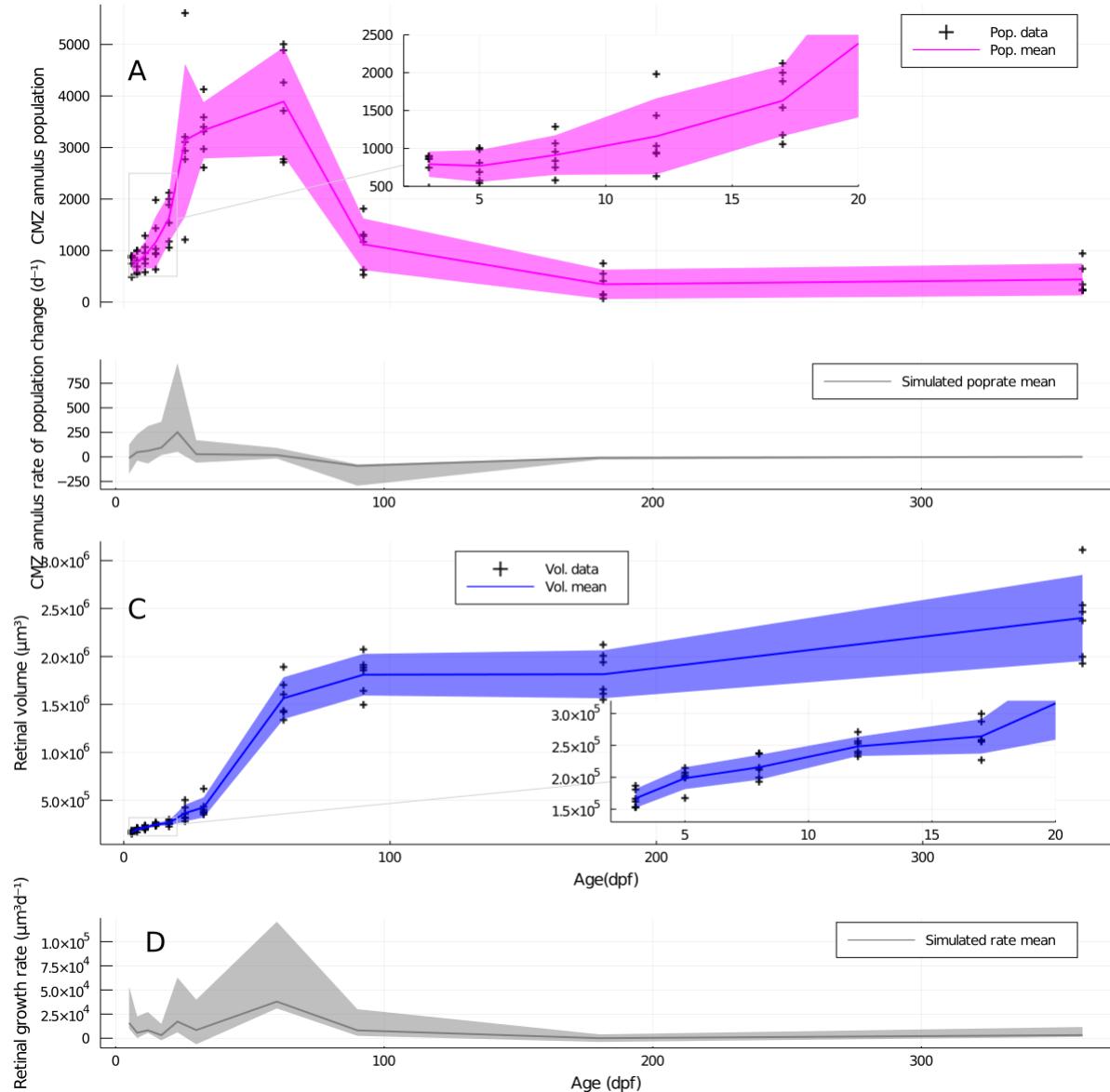


Figure 4.2: Population and activity of the CMZ over the first year of *D. rerio* life
 Panel A: Marginal posterior mean CMZ annulus population. Panel B: Marginal posterior mean retinal volume estimate. Insets in Panels A & B display data from 3-17 dpf. Panel C: Marginal posterior mean of the proliferative index of the CMZ annulus, assayed by specified retinal neurons with incorporated thymidine from an 8hr pulse at the indicated ages. Panel D: Mean daily rate of volumetric increase of the neural retina, calculated as the difference in volumes between two ages over the number of elapsed days. All means are displayed in a band representing the $\pm 95\%$ credible interval for the marginal posterior distribution of the mean.

is greater than the mean at 3dpf, which is a small negative value ($-5.5 \frac{cells}{d}$).

Subsequent to the bulk of the CMZ buildup and contribution to the retina, the population of the CMZ declines at about 39% the rate of its peak ascent, with an estimated $-90.1 \frac{cells}{d}$ by 90dpf. This process thins out the CMZ population to below its size immediately after embryogenesis, spread out

over a much larger peripheral annulus, for a much less dense mature CMZ. Interestingly, the 95% CI on the marginal posterior mean of estimated retinal volume at 180 dpf ($1.10 \pm 0.04e9 \mu\text{m}^3$) completely encompasses the 95% CI on volume at 90dpf ($1.11 \pm 0.03e9 \mu\text{m}^3$), while we assess a 99.6% probability of the 360dpf mean ($1.73 \pm 0.10e9 \mu\text{m}^3$) being greater than 180dpf. This suggests a possible second period of quiescence from approximately 90-180dpf, followed by steady contribution to the retina without a buildup in the CMZ population subsequently.

Given this rough description, it seems obvious that the ontogeny of the CMZ as a stem cell niche is characterised by different phases of activity, with different rates of proliferation and specification. It is not immediately clear what sort of periodization is justified by the data. It seems plausible that as few as two phases could explain the data well enough: an initial phase of logarithmic growth, with short cell cycle time and lower exit rate of RPCs from the CMZ into the specified neural retina, followed by a second phase of decay with longer cycle time and higher exit rate. On the other hand, perhaps some of the data features noted above justify a more bespoke model that captures, for instance, the initial quiescent period, or the post-180dpf growth of the retina. Because this is straightforwardly a question of how much model structure is justified by our data, we may address it as a model selection problem, using the system of Bayesian inference provided by nested sampling, and it is to this we now turn.

4.3 Periodization of postembryonic CMZ activity by Galilean Monte Carlo Nested Sampling

To perform our model selection task, we wish to simulate the time-evolution of CMZ population and retinal volume. This requires us to supply initial values for the size of the simulated CMZ populations and the volumes of the simulated retinae they are associated with. Given the findings presented in Figure 4.1, we are justified in initializing CMZ population and retinal volume by independent samples from the log-Normal models of their interindividual variability at 3dpf, at the end of embryogenesis and the beginning of CMZ-driven retinal growth. In order to produce new, simulated values of CMZ population and retinal volume, we apply a system of difference equations as follows, where pop_n is the population at n dpf, CT is the mean cell cycle time of the population in hours, and ϵ is the proportion of the population at time $n - 1$ that exits cycle and contributes to the volume of the specified neural retina, and μ_{cv} is the mean volume per cell contributed to the retina in μm^3 :

$$p_n = p_{n-1} \cdot 2^{\frac{24}{CT}} - p_{n-1} \cdot \epsilon \quad (4.1)$$

$$v_n = v_{n-1} + p_{n-1} \cdot \epsilon \cdot \mu_{cv} \quad (4.2)$$

A model "phase" can then be defined by the CT and ϵ parameters it applies to update the population and volume, over the appropriate number of days. The full parameterisation of a model with p phases is given by p pairs of CT and ϵ values, $p - 1$ phase transition times. μ_{cv} , which is taken to apply equally to all phases, is estimated from 3 dpf nuclear measurements, as described in ???. Given an initial population and volume sampled from the log-Normal models of their interindividual distributions, Equation 4.1 and Equation 4.2 may be applied to these values difference equations can be applied to produce simulated sample values at the times actually observed. Many such samples obtained by Monte Carlo can be used to estimate log-Normal distributions for the model parameters, which can be used to score the model

against observations. By defining prior distributions over the model parameters, we may sample from the prior to initialize a model ensemble. The ensemble can then be compressed by nested sampling, moving each model-particle over the parameter space by Galilean Monte Carlo, as described in ???. Using the typical procedures applied in nested sampling [Ski06], we estimated the Bayesian evidence, maximum a posteriori, and marginal posterior distributions on parameters for 2 and 3-phase models.

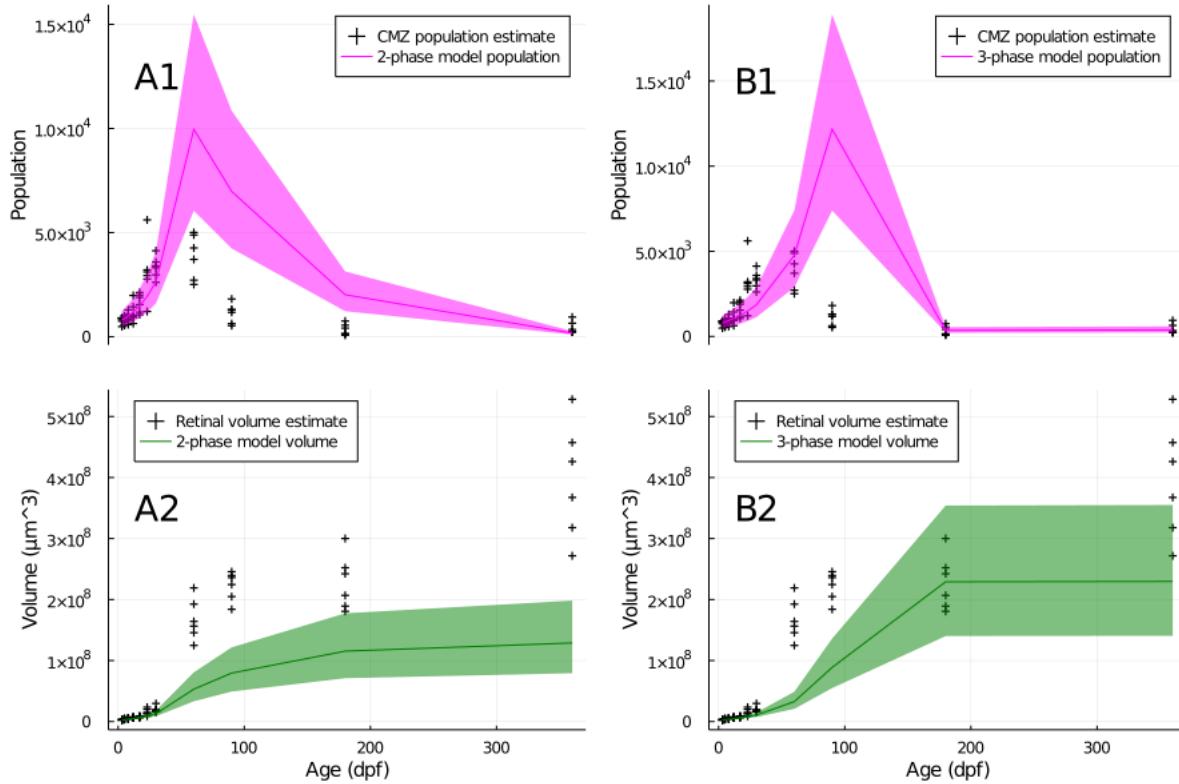


Figure 4.3: Maximum a posteriori output of periodization models

Population and volume estimates from observations (crosses) plotted with mean \pm 95% posterior mass model output, for the 2-phase model (A panels) and 3-phase model (B panels). A1,B1: population estimates. A2,B2: volume estimates.

It is useful to begin with the maximum a posteriori model output, as this plainly shows the primary problem with this simple model; the model relationship between changes in CMZ population and changes in volume breaks down after 30dpf. That is, the later volume estimates are too large for the CMZ to produce, given the calculated cellular volume at 3dpf. The result is that the models fit the early population and volume data quite well, but the population peak is dragged upward and forward to produce more-likely volume output. While it is possible that the later CMZ contributes more volume per neuron to the cellular retina, it is much more likely that the volume approximation applies better to more-nearly spherical eyes at younger ages, than to the flattened eyes of later ages. Although we had hoped that the estimated retinal volume data would constrain the ϵ exit rate parameters, this was not the case, as discussed below. When we tried floating μ_{cv} as a variable within the model, the MAP results were similar (data not shown), suggesting that a constant value for μ_{cv} across ages is the problem in achieving good model fits, not the particular value chosen. This reinforces the idea that the problem

relates to the breakdown of the retinal volume estimate at later ages.

While this limitation prevents either model from explaining the combined estimate datasets very well, they are in this sense under the same constraint, and so a reasonable inference about the number of phases justified by the data is still possible. Evidence estimates for the 2-phase and 3-phase models, given these data, are presented in Table 4.4. There are greater than 5600 orders of magnitude more evidence for the 2-phase model; this result has over 500 standard deviations of significance. This reflects the greatly expanded parameter space in the 3-phase model, which adds an additional 3 parameters to the 2-phase models' count of 5 parameters. The additional flexibility afforded by the 3rd phase in fitting the later volume data is unable to overcome the evidentiary penalty associated with the larger parameter space. We therefore conclude that, on the basis of these data, the 2-phase hypothesis must be accepted.

Table 4.4: Evidence favours a 2-phase periodization of CMZ activity

2-phase logZ	3-phase logZ	logZR	σ Significance
-5583.1 ± 4.9	-11247.0 ± 9.6	5664.0 ± 11.0	525.425

logZ: logarithm of $p(D)$, the marginal likelihood of the data, or model evidence. logZR: evidence ratio; positive ratios in favour of the 2-phase model. Largest evidence value bolded.

While the parameter estimates associated with these models are clearly unreliable, they are useful to inspect in order to demonstrate some properties of nested sampling, and for comparison to the simulations to follow. To begin with, we present the parameterization of the MAP models displayed above in Table 4.5. Unsurprisingly, the selected 2-phase model begins with a first phase of rapid proliferation, with a CT of 13.8 hr, followed by a second, slower phase of 27.2 hr. The imputed exit rate ϵ is greater than 200% of the day's starting population in the first phase, suggesting that new cells exit the CMZ after about 12 hours, around one cycle, while the second phase exit rate is much less, with 86% of the day's starting population exiting the CMZ, again suggesting a residency time of about one cycle. The imputed phase transition age is about 62dpf. Due to the volume estimate problem noted above, it is reasonable to believe that the CT estimates are likely too short, the ϵ estimates too high, and the transition age too late; all favoured in order to produce higher volume estimates at later ages.

Table 4.5: Maximum a posteriori parameter estimates for periodization models

Parameter	2-phase MAP	3-phase MAP
Phase 1 CT (h)	13.8	14.3
Phase 1 ϵ	2.28	2.18
Phase 2 CT (h)	27.2	35.4
Phase 2 ϵ	0.86	0.66
Phase 3 CT (h)	NA	109.1
Phase 3 ϵ	NA	0.12
Transition 1 age	61.9	115.5
Transition 2 age	NA	252.1

Because nested sampling naturally produces samples from the posterior, we can estimate posterior distributions using the evidence values for these samples. We performed this by kernel density estimation, specifically to investigate the extent to which the marginal posterior distributions for the selected model are polymodal; that is, the extent to which the evidence supports multiple hypotheses about

the parameters of the two imputed phases. Kernel density estimates (KDEs) for marginal posterior distributions on the 2-phase models' parameters are presented in Figure 4.4.

These estimates plainly reveal the polymodality of the posterior distributions, as well as the lack of constraint that the data impose on phase exit rates ϵ . For instance, the first phase parameters (top left) display 3 major modes; the best evidenced mode occurs between 15-20 hr CT , with the bulk of this posterior mass distributed above an exit rate of 1.0, but with a much larger range of ϵ supported than CT . The second phase parameters (bottom left) display even greater polymodality, with the bulk distributed around 140-150 hr CT , with an exit rate of less than .75. Plotting the two-dimensional marginal CT from both phases (top right) reveals that these parameters are the best constrained by the data; still, the posterior is highly polymodal, with numerous combinations of CT values receiving at least some support, although, obviously, shorter phase 1 CT values combined with longer phase 2 CT values are favoured. The marginal posterior distribution on the age at which the transition between phases occurs is plotted in the bottom right of Figure 4.4; while the largest peaks of the KDE are at times greater than 50dpf, a substantial portion of the prior mass falls earlier than this, significant for the simulations to follow.

It should be noted here that the maximum a priori model parameters are not perfectly reflected in the KDE. While the CT and ϵ parameters for both phases do fall broadly within the best-evidenced posterior modes, they are not centered therein; moreover, the MAP phase transition time does not occur in the best-evidenced transition time mode. As discussed in ??, an important property of nested sampling is that the accuracy of evidence calculations is traded off against the accuracy of estimating the posterior distribution. Since we have here prioritized evidence estimation, this is the cause of this discrepancy; the MAP models themselves have relatively little weight in the KDE estimates.

We conclude that, while our global model of CMZ population and volumetric retinal contribution is too badly flawed to make credible parameter estimates, a 2-phase model of this activity is far better substantiated by the evidence than a 3-phase model. We proceed on this basis, accepting the 2-phase model, and taking up the idea of the "slice model" introduced in Section 3.3.1, to investigate modelling the CMZ RPC population more concretely, directly from sectional observations, rather than from the calculated population and retinal volume estimates presented above.

4.4 Slice-model characterisation of asymmetrical CMZ population dynamics by Galilean Monte Carlo Nested Sampling

In the course of the preceding investigations, it became apparent that the population asymmetry mentioned in Chapter 3 was not a static phenomenon, with the dorsal lobe of the CMZ annulus being consistently more populous than the ventral lobe, as generally implied by the sources covered in Chapter 1. Rather, both the extent and orientation of asymmetry seem to evolve over time. Sectional population totals for the dorsal and ventral CMZ are presented in Figure 4.5, Panel A, alongside the related intra-individual asymmetry ratio in Panel B. The initially pronounced dorsal population and reduced ventral population both seem to go through the overall boom-bust progression of CMZ population, but their relative proportion within individuals reverses itself over the period from 17-90dpf. We also observed a similar phenomenon occurring across the naso-temporal axis over the same time period (Figure 11.1).

Inspected closely, these data provide a possible rationale for the reversal of asymmetry in the proliferative dynamics of the niche itself: the sectional (or "slice") population of the dorsal CMZ is increasing

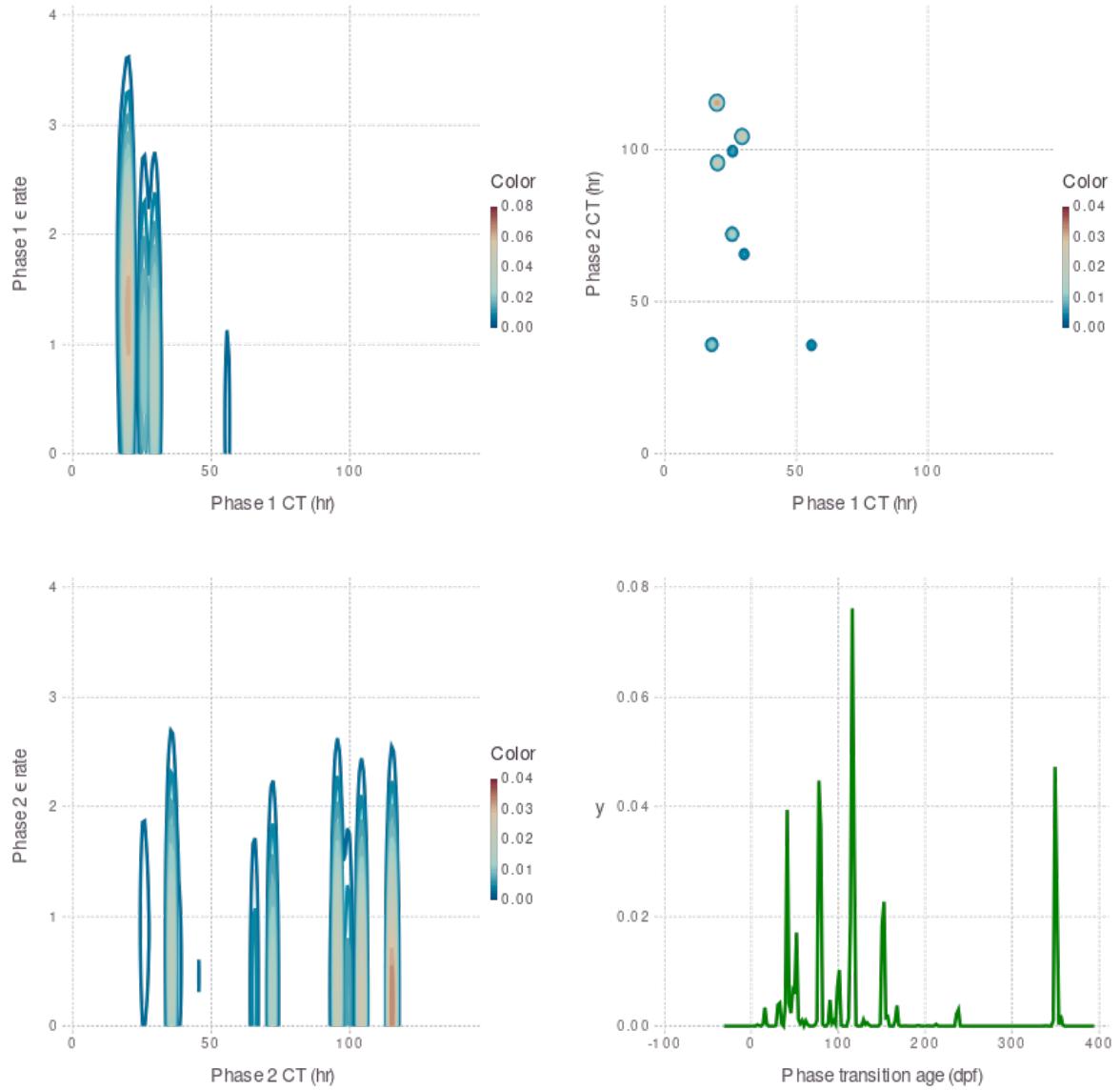


Figure 4.4: Kernel density estimates of marginal posterior parameter distributions, 2-phase model

beyond its postembryonic minimum by 12dpf, while the ventral CMZ takes until 17dpf to exhibit a noticeable increase in size; moreover, the peak dorsal population is achieved by 23dpf, whilst ventrally the peak is only achieved at 30dpf. This strongly suggests that the dorsal and ventral CMZ populations undergo similar, time-shifted processes of proliferation from different starting populations. If this is so, an explanation for this time-shifted phenomenon could have fundamental relevance to predicting and controlling the proliferative behaviour of peripheral RPCs and stem cells.

To test this hypothesis, we used a “slice model” of the CMZ, where the thickness of the slice is taken to be the same as the observed cryosection thickness ($14\text{ }\mu\text{m}$). The population of the CMZ is modelled with a difference equation, as above, but with an additional exit term representing lateral, circumferential

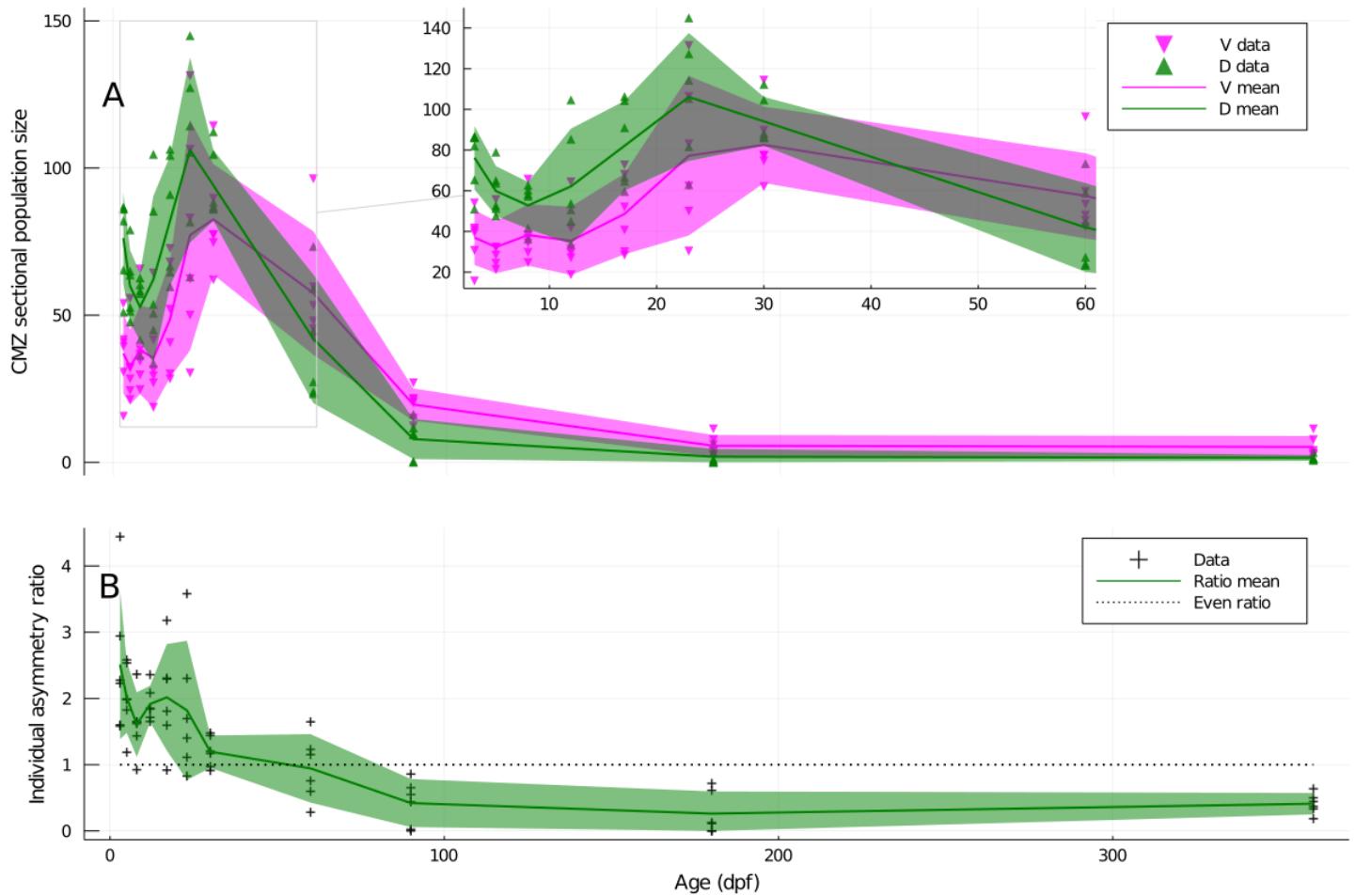


Figure 4.5: Developmental progression of dorso-ventral population asymmetry in the CMZ. Marginal posterior distribution of mean dorsal (D) and ventral (V) population size in $14\mu\text{m}$ coronal cryosections (panel A) or intra-individual D/V count asymmetry ratio (panel B), $\pm 95\%$ credible interval, $n=5$ animals per age. Data points represent mean counts from three central sections of an experimental animal's eye.

contributions of the CMZ to the generation of new, adjacent “slices”. The value of this term is calculated from the difference in CMZ annulus diameter over the calculated time period implied by a power-law model of lens growth fitted to observations, as discussed in Section 10.1.4. The resultant difference equation is Equation 4.3. Terms are as defined above, except that p_n is the sectional population at n dpf, and not the total CMZ annulus population; additionally, η is defined as the daily circumferential exit rate implied by the power-law model.

$$p_n = p_{n-1} \cdot 2^{\frac{24}{CT}} - p_{n-1} \cdot \epsilon - \eta \quad (4.3)$$

We reasoned that, if the phase transition occurs earlier in the dorsal CMZ than in the ventral CMZ, there should be some informational gain in separating these observations vs. a combined total sum for both lobes of the slice annulus⁴. In this case, the combination of two phase-shifted populations in the

⁴The “total” model is required to supply double the circumferential exit rate of the dorsal or ventral models, to reflect

total model should produce a “fuzzy” peak relative to the separate modes. We therefore estimated the evidence, MAP, and posterior marginals for 2-phase models given the sectional sum, dorsal, and ventral populations. Given our belief that the whole-eye model presented above gives an over-long transition time, and in order to focus on the most informative subset of the data for our hypothesis, we restricted this analysis to the population data within the first three months of life.

The slice model proves to have much greater success at explaining sectional counts than the whole-eye model does at explaining the annulus population estimates; maximum a posteriori model output is presented in Figure 4.6. In particular, all of the models adequately represent the early decline in sectional populations, arising from rapid early growth of the eye that exceeds the CMZs’ proliferative capacity, without further ado; if there is not sufficient evidence to justify a separate, slow, late phase of proliferation, there is clearly none to justify a separate, slow, early phase.

The hypothesis that there is a time-shift in the phase transition across the dorso-ventral axis is thoroughly refuted by these models, in two ways. First, the evidence estimates demonstrate that we are not epistemically justified in separating the dorsal and ventral populations. The total-population slice model receives greater than 500 orders of magnitude more evidence than the joint evidence for the separate dorsal and ventral models, with greater than 190 standard deviations of significance, as displayed in Table 4.6. It is interesting to note that the evidence for the dorsal model is substantially lower than either the total or ventral models. This may indicate a causal influence on the dorsal population that is neither in the model nor acting on the ventral population, or it may be an uninteresting sampling effect.

Table 4.6: Evidence favours a combined slice model over separate dorsal and ventral models

Total logZ	Dorsal logZ	Ventral logZ	logZR	σ Significance
-498.1 ± 1.6	-638.3 ± 1.8	-371.7 ± 1.0	512.0 ± 2.6	194.861

logZ: logarithm of $p(D)$, the marginal likelihood of the data, or model evidence. logZR: evidence ratio; positive ratio in favour of the combined model. Largest evidence value bolded.

A second line of evidence indicating that this hypothesis is unsupported are the MAP model parameter values, summarized in Table 4.7. The MAP phase transition age for the total slice model is functionally identical to that for the ventral model, and within two days of the MAP transition for the dorsal model. Additionally, the dorsal MAP transition is actually later than the ventral date, which further suggests the original idea of a time-shifted late ventral phase change has no evidence. Reassuringly, the MAP parameters for the total slice model are similar to those found for the 2-phase MAP in Figure 4.4. Interestingly, the MAP phase parameters differ markedly between the split dorsalventral models and the total slice model, although all suggest a markedly longer CT and lower ϵ for the second phase. This observation suggests that the additional noise incurred from splitting the sectional CMZ population into dorsal and ventral lobes has a strong effect on parameter estimates. This may provide a practical reason to prefer combining these counts in slice models, beyond the fact that it is epistemically unjustified on the support of these data.

Examining the marginal posterior distributions of the total slice model, presented in Figure 4.7, shows that exit rate posteriors are no less constrained than the whole-eye model, suggesting the retinal volume estimate supplies little additional information, beyond the population estimate, regarding the rate at which RPCs leave the niche. However, the marginal posteriors on cycle length are less constrained than

the requirement for this “full” slice to grow both lobes of the eye

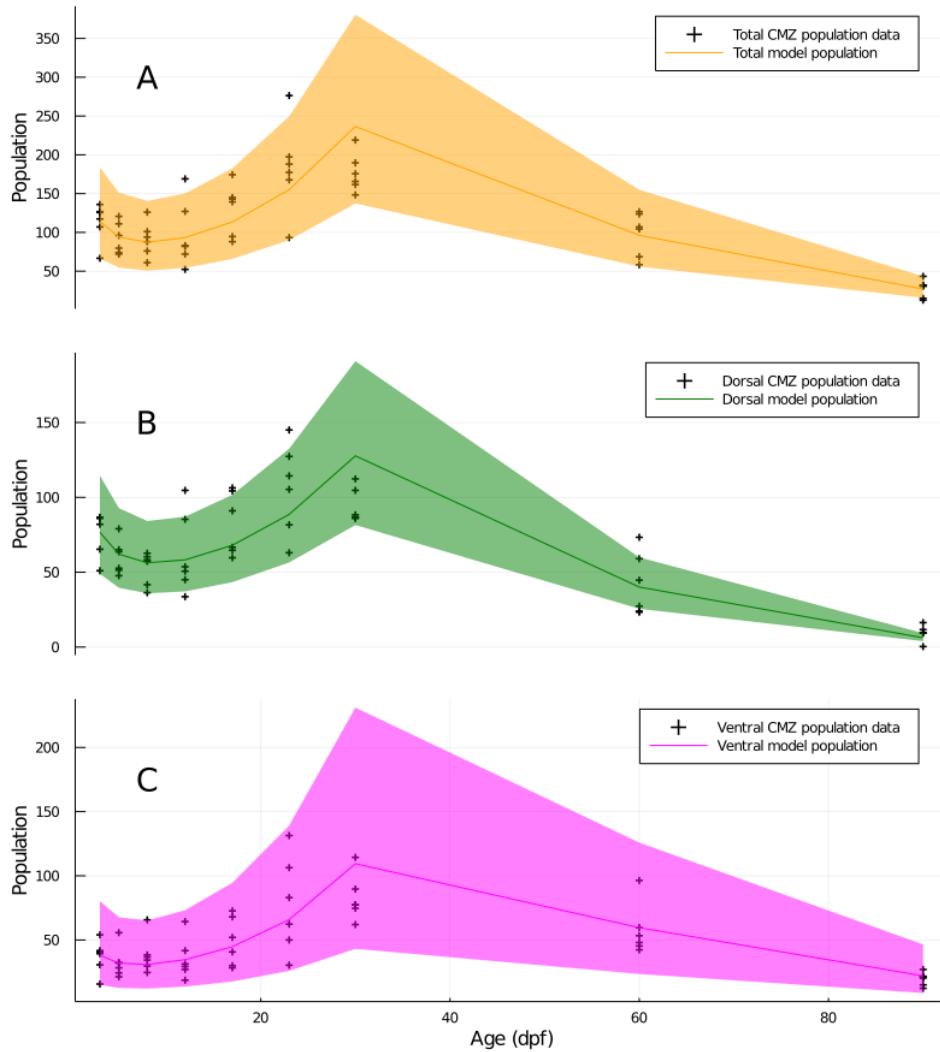


Figure 4.6: Maximum a posteriori output of total, dorsal, and ventral CMZ slice models
Population and volume estimates from observations (crosses) plotted with mean \pm 95% posterior mass model output, for the total CMZ population model (panel A), dorsal CMZ population model (panel B), and ventral CMZ population model (panel C).

Table 4.7: Maximum a posteriori parameter estimates for slice models

Parameter	Total MAP	Dorsal MAP	Ventral MAP
Phase 1 CT (h)	16.1	29.1	20.5
Phase 1 ϵ	1.72	0.69	1.15
Phase 2 CT (h)	23.5	37.0	55.2
Phase 2 ϵ	1.06	0.62	0.38
Transition age	35.5	37.2	35.8

the whole-eye model, with noticeably less separation between modes, which suggests that the volume estimate may nonetheless narrow the range of credible CT values. This suggests a rationale for pursuing

better volumetric estimates for retinae from fish older than 30 dpf. The posterior distribution on the age of phase transition indicates that many values for this parameter retain some credibility, and, like the whole-eye model marginals in Figure 4.4, the MAP estimate does not occupy the most probable KDE kernel for this value, for the same reason noted above.

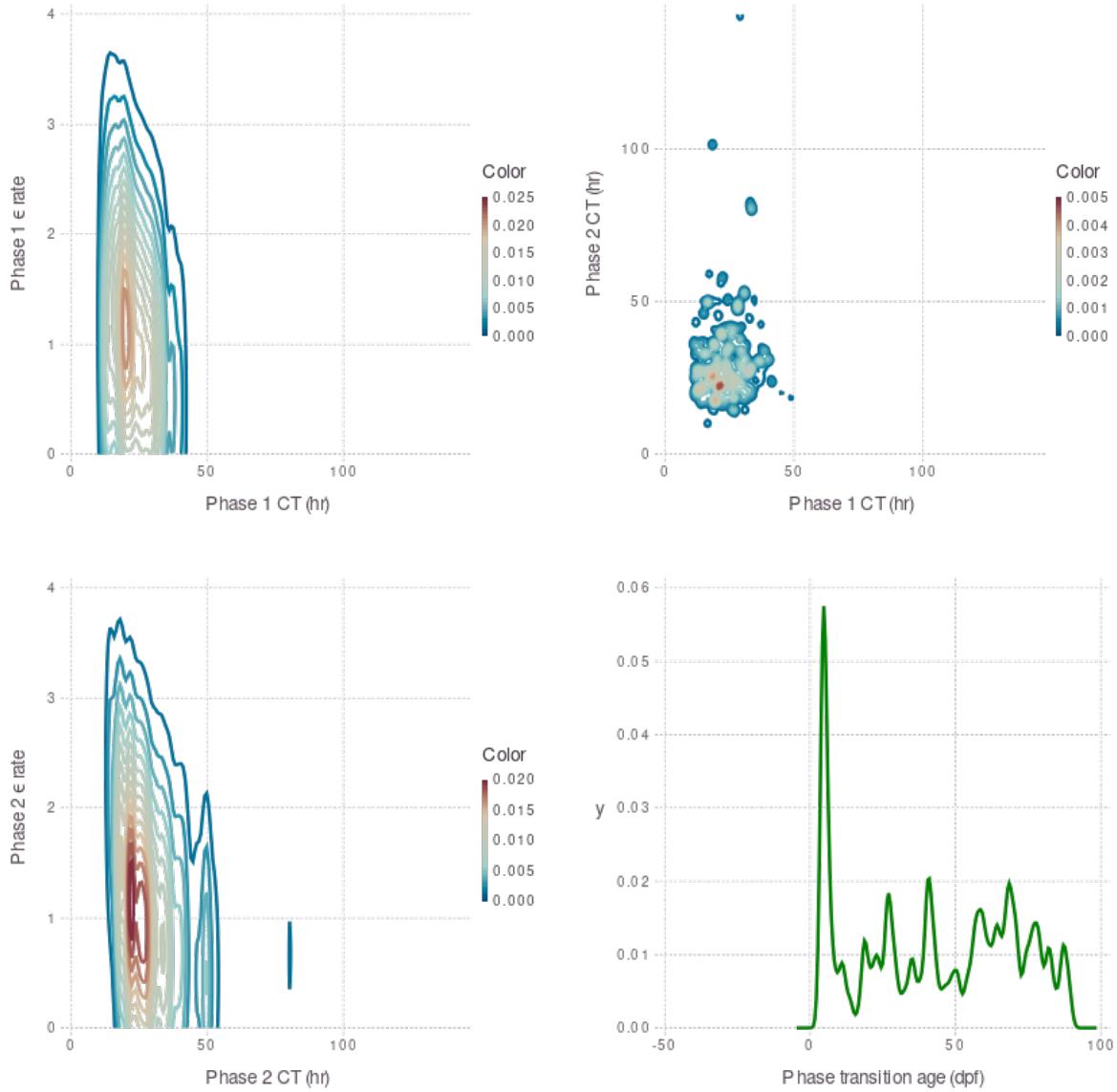


Figure 4.7: Kernel density estimates of marginal posterior parameter distributions, total slice model

On the basis of this analysis, the best available hypothesis about the observed ontogeny of RPC population asymmetry across the dorsoventral axis is the pre-establishment of differential population size by asymmetrical progress of the embryonic specification wave, rather than differential proliferative schedules.

4.4.1 Cumulative thymidine labelling supports GMC-NS model predictions

The similarity of CT and ϵ exit rate parameters in the whole-eye and total slice models, despite the defective cellular retinal volume estimate, suggests that these values may be relatively accurate. In order to partially validate this idea, we examined 3dpf CMZ RPCs labelled with a 10.5 hour pulse of the thymidine analogue EdU. We used the Empirical Bayes approach to estimate the evidence for separate Nowakowski-style [NLM89] linear models for the dorsal and ventral CMZ, against a model for both subpopulations combined. While this model is inadequate for reasons described in ??, it can serve to substantiate the first phase cycle time parameter. These results are summarized in Table 4.8, with the relevant linear regressions displayed in Supplementary Figure 11.2.

Table 4.8: Evidence favours whole-CMZ linear cycle models over separate D/V models

Model	Implied T_c (hr)	Implied T_s (hr)	logZ
Dorsal	14.7 ± 1.6	1.38 ± 0.76	7.778
Ventral	14.0 ± 1.2	0.8 ± 0.58	15.202
Combined	14.6 ± 1.1	1.25 ± 0.53	26.165

T_c : calculated cell cycle time. T_s : calculated s-phase length. logZ: logarithm of p(D), the marginal likelihood of the data, or model evidence. Largest evidence value bolded.

Firstly, there are approximately 3 orders of magnitude of evidence in favour of the combined model. There is no evidence, on this basis, for differing cell cycle characteristics across the D/V retinal axis of asymmetry at 3dpf. Moreover, the Nowakowski-calculated cell cycle time T_c for the combined model, 14.6 ± 1.1 hr, includes the whole-eye MAP first phase TC , 13.8 hr, within one standard deviation, while the total slice-model first phase TC , 16.1, is within two. This confirms that the differing D/V MAP parameter estimates observed in the slice model are unsubstantiated; the evidence supports broadly similar cell cycle characteristics across anatomical axes in 3dpf CMZ RPCs. Moreover, at least the MAP cell cycle times implied by our model selection time seem realistic, given the assumptions of the of Nowakowski model. That said, the data clearly diverge from a linear trend toward the end of the pulse, and the calculated S phase lengths are unrealistically short, showing the limitations of this model.

4.4.2 Asymmetry in dorso-ventral cohort contributions highlights exit rate parameter slop

4.5 Investigating CMZ RPC lineage outcomes

By labelling CMZ RPCs with the thymidine analogue EdU in an 8 hour pulse at 3, 23, and 90 dpf, followed by histochemical analysis for known zebrafish retinal neural lineage markers, we investigated the possibility that RPC lineage outcomes change over the life of the organism. This hypothesis is of particular interest, as differences in the mosaic organisation of embryonically-contributed central retina and CMZ-contributed peripheral retina remain unexplained [ABS⁺10]. It may, moreover, have clinical significance, were quiescent peripheral stem cells to be entrained for regenerative medical purposes, as their lineage outcomes are may be different than embryonic RPCs. We used antibodies raised against Pax6 and Isl2b to mark retinal ganglion cells (RGCs) of the ganglion cell layer and amacrine cells of the inner nuclear layer. Anti-glutamine synthetase (GS) and anti-PKC β were used to mark Müller glia (MG)

and bipolar cell (BPC) populations of the INL. The unique flattened nuclear morphology of horizontal neurons was used to identify them. Lastly, the antibody Zpr1, directed against an unknown antigen present in photoreceptors with double cone morphology, was used to mark these cells.

Observations were collected in "staining groups", which combined histological markers; representative confocal micrographs from this study in animals pulsed at 23dpf are displayed in Figure 4.8, while data from all ages are plotted in Figure 4.9. It is worth noting that the relative position of the CMZ-contributed cohort is very different in older animals, with 7 days of chase time being just enough for the majority of the 90dpf cohort to be reliably located within the specified neural retina, as depicted in supplementary ???. Because we suspect that CMZ-contributed retinal cohorts are subject to a process of attrition, and that this turnover might be higher near the CMZ, as documented in Section 4.6, if this hypothetical turnover process has differential effects on specified retinal neurons of different lineages, results may not be directly comparable between ages. With that caveat, we proceed to the lineage data.

These data take the form of fractions of the presumptive CMZ-contributed, thymidine-labelled cohort entering each of the three cellular layers (panel A of Figure 4.9), or the subfraction of the cohort within a given layer expressing a particular cellular marker (Panels B-I). While some variability is apparent in all of the measurements, it is unclear whether it is well-described as time-dependent in most cases. In order to address the question of whether lineage outcomes differ over time, we needed to assess the joint evidence for separate models of each measurement at each age, against the evidence for a single model of the measurement for all of the assessed ages.

Because it is not immediately obvious that these fractional measurements are better described log-Normally as the underlying population counts are, we first assessed the joint evidence for Normal and log-Normal models of the data, summarized in Supplementary Section 11.4. Our evidence supports Normal modelling of all measurements aside from PKC β -, GCL Pax6⁵, and GS-positive cells, along with those displaying horizontal nuclear morphology. We noted that the evidence estimates contradicted simple likelihood ratio tests in the cases of INL Pax6-, and Zpr1-positive cells, Isl/Pax6-double positives, and for the fraction of the cohort contributed to the ONL, summarized in Supplementary Section 11.5. This demonstrates a case where full evidence estimation produces the opposite result from maximum likelihood methods for multiple separate measurements. Without a clear fundamental justification for uniformly preferring one model or the other, we selected the best-supported model for each measurement. We estimated the evidence for an age-marginalized model, representing stable contribution to the layer or lineage over time, and compared this to the joint evidence for separate models at each age, representing a time-varying contribution model. These estimates are presented in Table 4.9.

Generally speaking, the evidence speaks to the overwhelming compositional stability of retinal contributions across this period. The stable model is supported over time-varying models by significances of greater than 10 standard deviations for all layers and lineages except Pax6-positive cells in the GCL and INL, presumptive amacrine neurons. In these cases, the evidence is marginally in favour of a stable model in the GCL, but with only two standard deviations significance, and is ambiguous in the INL, with a marginal but not significant evidentiary advantage for the time-varying model. Therefore, our lineage tracing measurements provide no clear evidence for the periodization of CMZ retinal contributions that would accompany the observed periodisation of proliferative and specificative behaviours. It is possible that a subtle periodisation of amacrine production exists; this is the only measurement where more data might plausibly alter these initial conclusions.

⁵Pax6-positive fractional lineage contributions to the INL support Normal models better than Log-Normal.

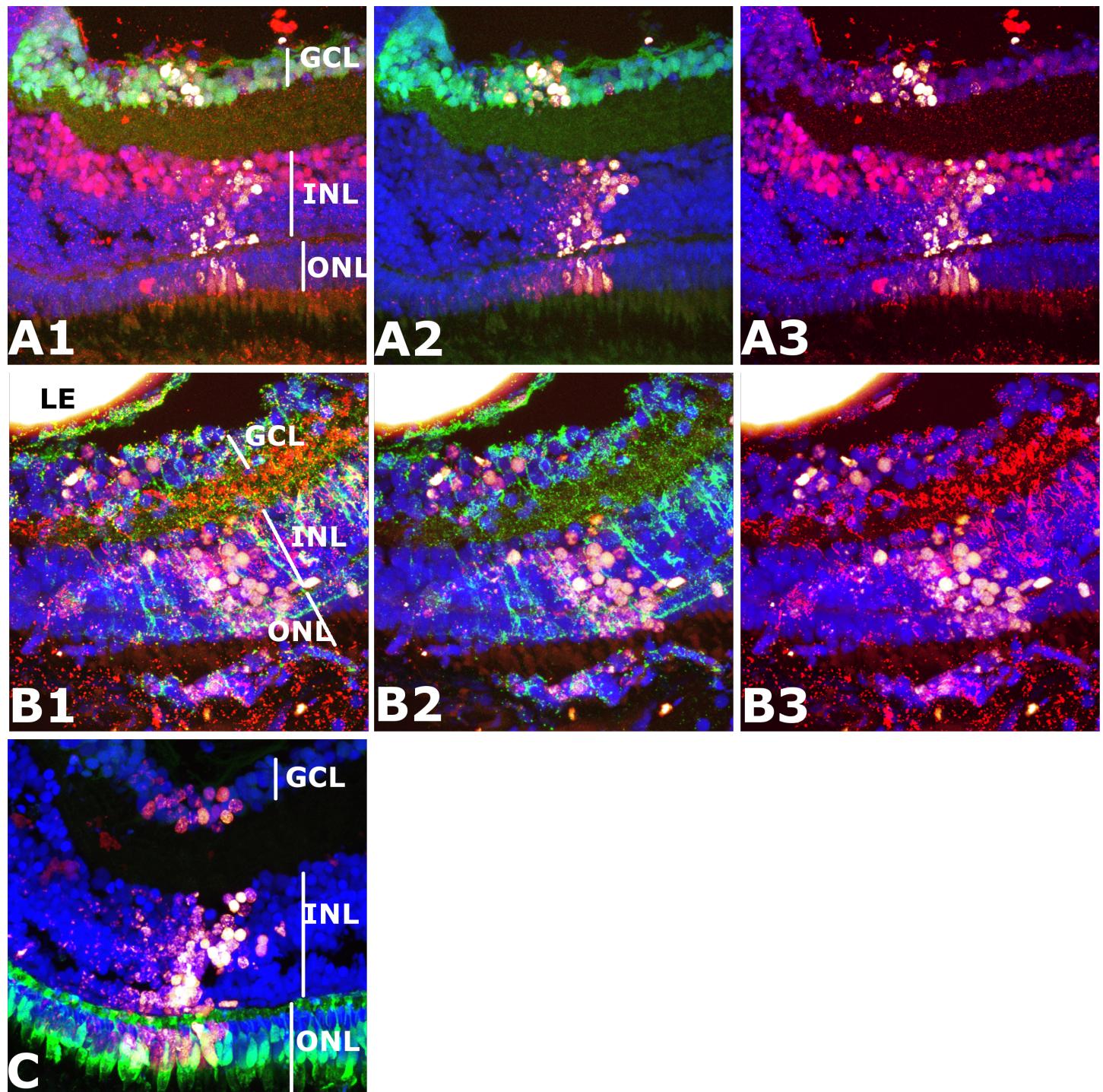


Figure 4.8: **Representative 23dpf lineage marker confocal micrographs**
Panel A1: RGC/Amacrine staining group. A2: Isl2b channel. A3: Pax6 channel.
Panel B1: MG/BPC staining group. B2: PKC β channel. B3: GS channel.
Panel C: Double cone staining group. Zpr1 channel.
GCL: Ganglion cell layer. INL: Inner nuclear layer. ONL: Outer nuclear layer.

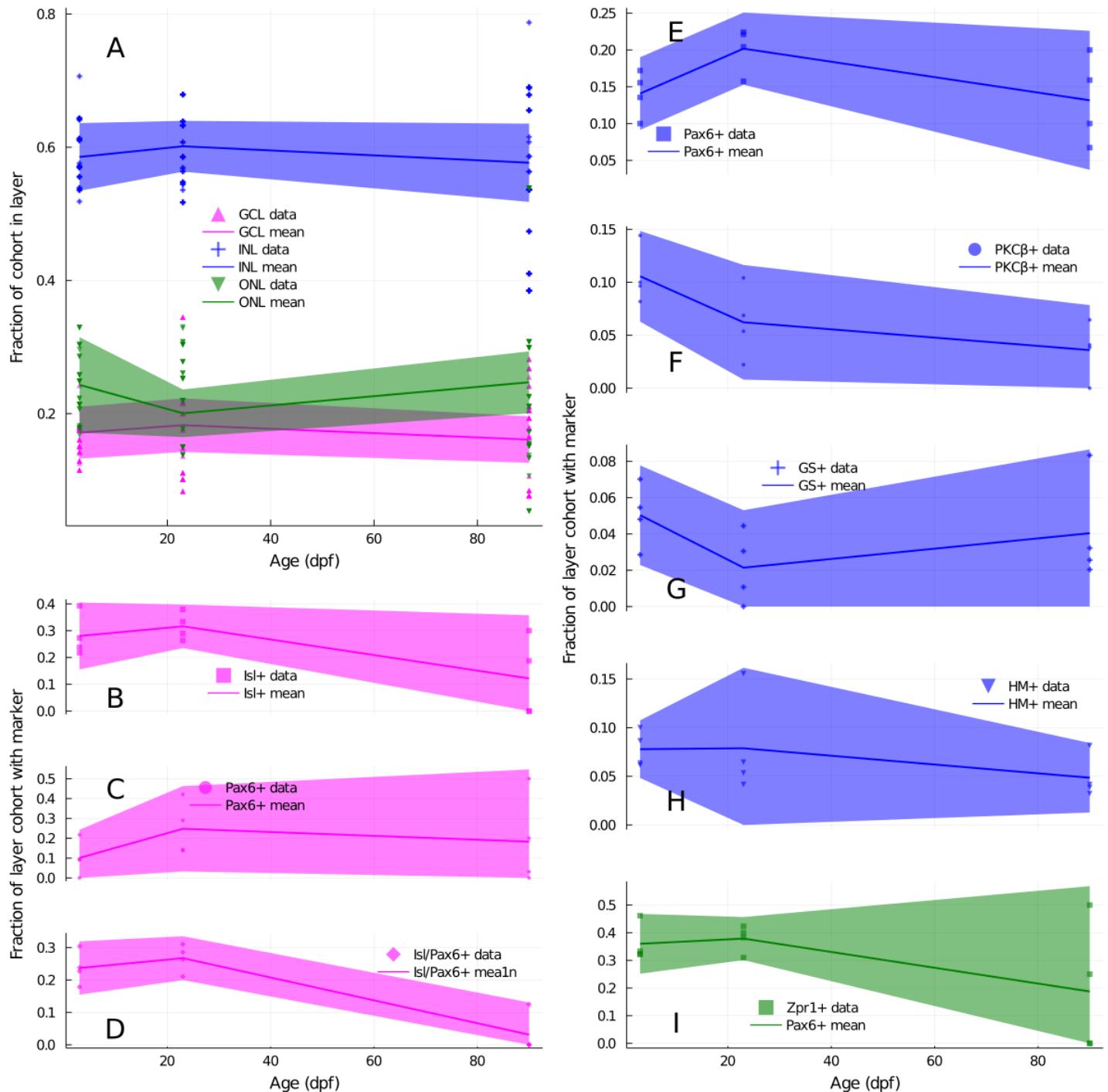


Figure 4.9: CMZ contributions to the neural retina over time by layer and lineage marker
 Marginal posterior distribution of mean dorsal (D) and ventral (V) population size in $14\mu\text{m}$ coronal cryosections (panel A) or intra-individual D/V count asymmetry ratio (panel B), $\pm 95\%$ credible interval, $n=5$ animals per age. Data points represent mean counts from three central sections of an experimental animal's eye.

Table 4.9: Evidence supports stable model of retinal contributions

Layer	Marker	Cell type	Stable logZ	Time-vary logZ	logZR	σ sign.
GCL	Cohort	All GCL cells	-154.9 ± 1.1	-189.0 ± 1.4	34.1 ± 1.8	19.1
GCL	Isl2b	RGC	-34.08 ± 0.55	-74.5 ± 1.0	40.4 ± 1.2	34.4
GCL	Pax6	Displaced am.	-77.327 ± 0.041	-77.87 ± 0.27	0.54 ± 0.27	2.0
GCL	Isl2b/Pax6	RGC subtype	-45.61 ± 0.76	-97.9 ± 1.0	52.3 ± 1.3	41.
INL	Cohort	All INL cells	-89.77 ± 0.78	-184.0 ± 1.4	94.3 ± 1.6	57.3
INL	Pax6	Amacrine cell	-37.11 ± 0.47	-36.83 ± 0.88	-0.3 ± 1.0	0.3
INL	PKC β	Bipolar cell	9.76 ± 0.25	-0.77 ± 0.67	10.53 ± 0.71	14.8
INL	GS	Müller glia	21.23 ± 0.65	6.7 ± 1.1	14.5 ± 1.3	11.6
INL	HM	Horizontal cell	23.21 ± 0.32	6.73 ± 0.61	16.48 ± 0.68	24.1
ONL	Cohort	All ONL cells	-113.78 ± 0.78	-201.0 ± 1.5	87.3 ± 1.7	50.8
ONL	Zpr1	Double cones	-35.37 ± 0.6	-93.7 ± 1.4	58.3 ± 1.5	38.6

logZ: logarithm of p(D), the marginal likelihood of the data, or model evidence. Largest evidence values bolded. logZR: evidence ratio; positive values in favour of stable model.

4.6 Microglial apoptotic fate of *D. rerio* retinal neurons & functional significance of CMZ activity

Recently, extensive neural death has been reported in older zebrafish retinae, described as a "neurodegenerative pathology" and suggested as a model of age-related neurodegeneration [VGV⁺18]. In the course of our thymidine analogue pulse-chase studies, we noted that it often appeared that CMZ-contributed cohorts had been "thinned out" noticeably only a month or two after their entry into the neural retina, even in juveniles of 30-90 days of age. If neural retinal turnover is a general phenomenon throughout the life of the organism, this has fundamental implications for the view that this phenomenon should be treated as pathological, rather than constitutive. However, the thinning phenomenon could be explained by processes involved in the changing morphology and geometry of the neural retina during this period. In particular, the neural retina thickens noticeably over this time period, as displayed in supplementary ???. Although this increase is due in large part to the lengthening of photoreceptor outer segments, the inner nuclear layer is also significantly thickened. It is plausible that, for instance, labelled CMZ cohorts are "invaded" by unlabelled neighbours during this process, giving the appearance of "thinning" without its quantitative reality.

In order to investigate this phenomenon, we administered 24hr pulses of EdU to 1dpf embryo, and followed with a 24 hr pulses of BrdU at 23dpf to mark a CMZ-contributed cohort near the height of its activity. By taking both coronal and transverse sections through animals at 30, 60, and 90 dpf, we sampled these cohorts from both morphological axes of the retina and counted labelled sectional totals.

Chapter 5

Mutant npat results in nucleosome positioning defects in *D. rerio* CMZ progenitors, blocking specification but not proliferation

5.1 Introduction

The zebrafish (*D. rerio*) circumferential marginal zone (CMZ), located in the retinal periphery, contains the retinal stem cells and progenitors responsible for the lifelong retinal neurogenesis observed in this cyprinid. Analogous to CMZs in other model organisms, such as *X. laevis* [PKVH98], it has been of particular interest to us since the discovery of quiescent stem cells at the mammalian retinal periphery [Tro00], as an understanding of the molecular mechanisms regulating this proliferative zone may shed light on whether these mammalian cells might be harnessed for the purpose of regenerative retinal medicine. While significant progress has been made in this direction [RBPP06], molecular lesions in a plethora of zebrafish mutants displaying defects in CMZ development and activity remain largely uncharacterised. We examine here one such microphthalmic line identified in an ENU screen, *rys* [WSM⁺05], characterised by Wehman et al. as a Class IIA CMZ mutant, with a small eyes (see Figure 5.1) and apparently paradoxically enlarged CMZ.

Mapping revealed the causative *rys* mutation lay in the zebrafish npat gene, the nuclear protein associated with the ataxia-telangiectasia locus in mammals [IYS⁺96]. Although npat is heretofore uncharacterised in zebrafish, its mammalian homologues, human NPAT and mouse Npat, have been extensively examined. These studies have demonstrated that NPAT plays a critical role in coordinating events associated with the G1/S phase transition in proliferating cells [YWNH03]. S-phase entry requires tight co-ordination between the onset of genomic DNA synthesis and histone production, in order to achieve normal chromatin packaging and assembly. NPAT, found in the nucleus [STN⁺02] and localised, in a cell-cycle dependent manner, to histone locus bodies [GDL⁺09], induces S-phase entry [ZDI⁺98] and activates replication-dependent histone gene transcription by direct interaction with histone gene clusters [ZKL⁺00] in association with histone nuclear factor P (HiNF-P) [MXM⁺03]. The protein's effects

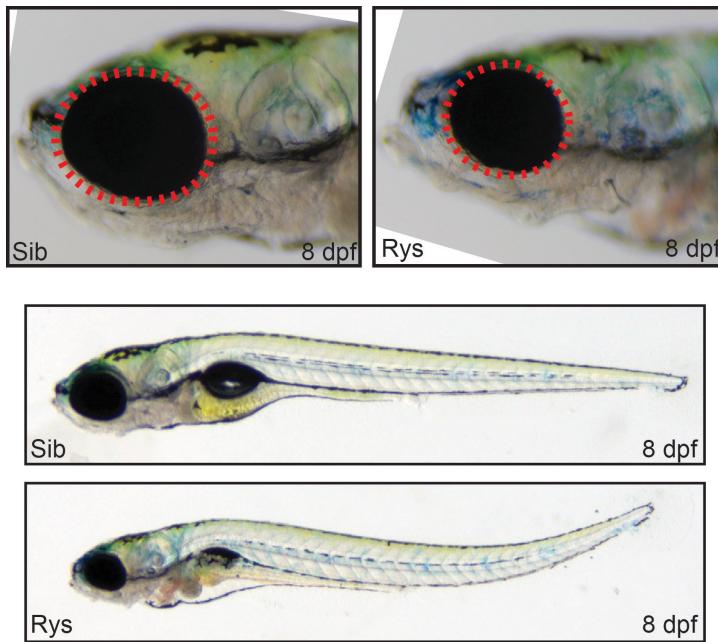


Figure 5.1: *rys* mutants exhibit a small-eye phenotype

on S-phase entry and histone transcription are associated with distinct domains at the C-terminus and N-terminus, respectively [WJH03]. NPAT is also known to associate with the histone acetyltransferase CBP/p300 [WIEO04] and directs histone acetylation by this enzyme [HYSL11].

NPAT is known to be a component of the E2F transcriptional program [GBB⁺03] and a substrate of cyclin E/CDK2 [ZDI⁺98], although E2F-independent activation by cyclin D2/CDK4 in human ES cells has also been described [BGL⁺10]. The expression of NPAT protein peaks at the G1/S boundary [ZDI⁺98], as does its phosphorylation, which promotes its transcriptional activation of replication-dependent H2B [Ma00] and H4 [MGv⁺09] genes, while its effect on low, basal levels of H4 transcription is phosphorylation-independent [YWNH03]. Of particular interest, NPAT has recently been found to be required for CDK9 recruitment to replication-dependent histone genes [PJ10]; CDK9 and monoubiquitinated H2B are essential for proper 3' end processing of stem-loop histone transcripts [PSS⁺09]. NPAT and HiNF-P have also been found associated with the U7 snRNP complexes that perform this function [GDL⁺09]. The replication-dependent activities of NPAT are thought to be terminated by WEE1 phosphorylation of H2B, which excludes NPAT from histone clusters [MFKM12].

All of these studies have been conducted in tissue culture contexts, perhaps due to the challenges associated with studying this critical protein *in vivo*; mouse embryos with provirally inactivated Npat arrest at the 8-cell stage, for instance [DFWV⁺97]. The availability of *rys*, a zebrafish npat mutant which develops well into the larval stage, is therefore of considerable interest, as it allows for the study of npat's function within complete tissues. We demonstrate here that npat is critical for the normal proliferation and differentiation of CMZ neural progenitors.

The altered npat regulation of histone transcription and cell cycle progression in *rys* results in a dramatically shortened S-phase and an increase in the abundance of core histone transcripts, including polyadenylated transcripts, and their associated proteins. These effects are associated with disorganized nucleosome positioning and altered protein expression and progenitor identity, failure of *rys* CMZ cells

to contribute to the neural retina, and subsequent cell death, suggesting that npat may play a critical role in the coordination of genomic events required for normal execution of differentiation programmes.

5.2 Results

5.2.1 The *rys* CMZ phenotype is characterised failure of RPCs to specify, altered nuclear morphology, aberrant proliferation and expanded early progenitor identity

rys has previously been described as having an enlarged CMZ [WSM⁺⁰⁵], but whether this is a consequence of an enlarged proliferative population, altered cellular morphology, or something else, remained unclear. We sought to learn more about the ontogeny of the mutant niche by examining two histochemical markers of cycle activity during the early life of *rys*; Proliferating Cell Nuclear Antigen (PCNA), which in *D. rerio* is present throughout the cell cycle, and the genomic incorporation of the thymidine analogue EdU over a 24 hour pulse, which indelibly marks cells that have passed through S-phase whilst exposed to it. These data are displayed in Figure 5.2. During this period, the PCNA +ve population of the sibling CMZ declines precipitously (Panel A), with a corresponding decrease in proliferative activity as measured by the incorporation of EdU (Panel B). Whether or not the *rys* CMZ population is enlarged by comparison depends on the age at which it is sampled, with 99.6% of the marginal posterior mass of the estimated mean sectional *rys* CMZ PCNA +ve population below the sib mean at 4dpf, while 99.8% of the marginal posterior of the 10dpf *rys* mean lies above that of sibs. Therefore, the *rys* CMZ population is better described as achieving its peak periembryonic size later than siblings; its population is only numerically larger than sibs at later ages¹.

¹*rys* animals universally die by approximately 3 weeks of age.

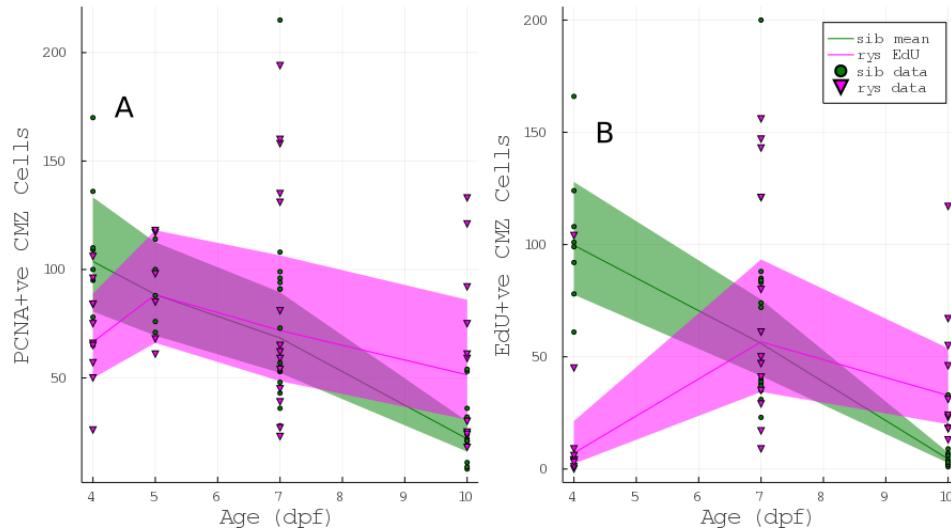


Figure 5.2: *rys* CMZ populations start relatively small and quiescent, end abberantly large and proliferative

Panel A: Counts and estimated mean $\pm 95\%$ credible intervals of PCNA-positive cells in the peripheral CMZ of *rys* (magenta) and their siblings (green). Panel B: As above, but for counts of double PCNA-, EdU-positive cells in the CMZ after a 24 hour pulse of EdU.

Surprisingly, very few *rys* animals have many actively cycling RPCs at 4dpf, by comparison to the robustly cycling sib RPCs at this age; a mean estimate of $96.2 \pm 3.4\%$ of sib RPCs are labelled during the 24 hr pulse at 4dpf, while only $24.1 \pm 37.2\%$ of *rys* RPCs are. The situation is broadly reversed at 10dpf, with only $24.4 \pm 14.9\%$ of sib RPCs labelled, compared to $65.8 \pm 16.2\%$ of *rys* RPCs. While we questioned whether this late-stage uptick in *rys* labelling might not represent actual mitotic activity, we were able to readily find mitotic figures within these populations, shown in supplementary Figure 12.2. As the data convey, these outcomes are highly variable in both sib and *rys* animals, with, for instance, an isolated mutant at 4dpf sporting an EdU-positive population near the sibling mean. However, even in those *rys* animals which do have actively proliferating RPCs at these earlier ages, there is a marked failure of the CMZ to contribute to the postmitotic, specified neural retina. Confocal micrographs displaying *rys* CMZ cohorts labelled with BrdU at 3dpf that have failed to enter the neural retina after 7 days of chase time are displayed alongside their normal siblings in Figure 5.3.

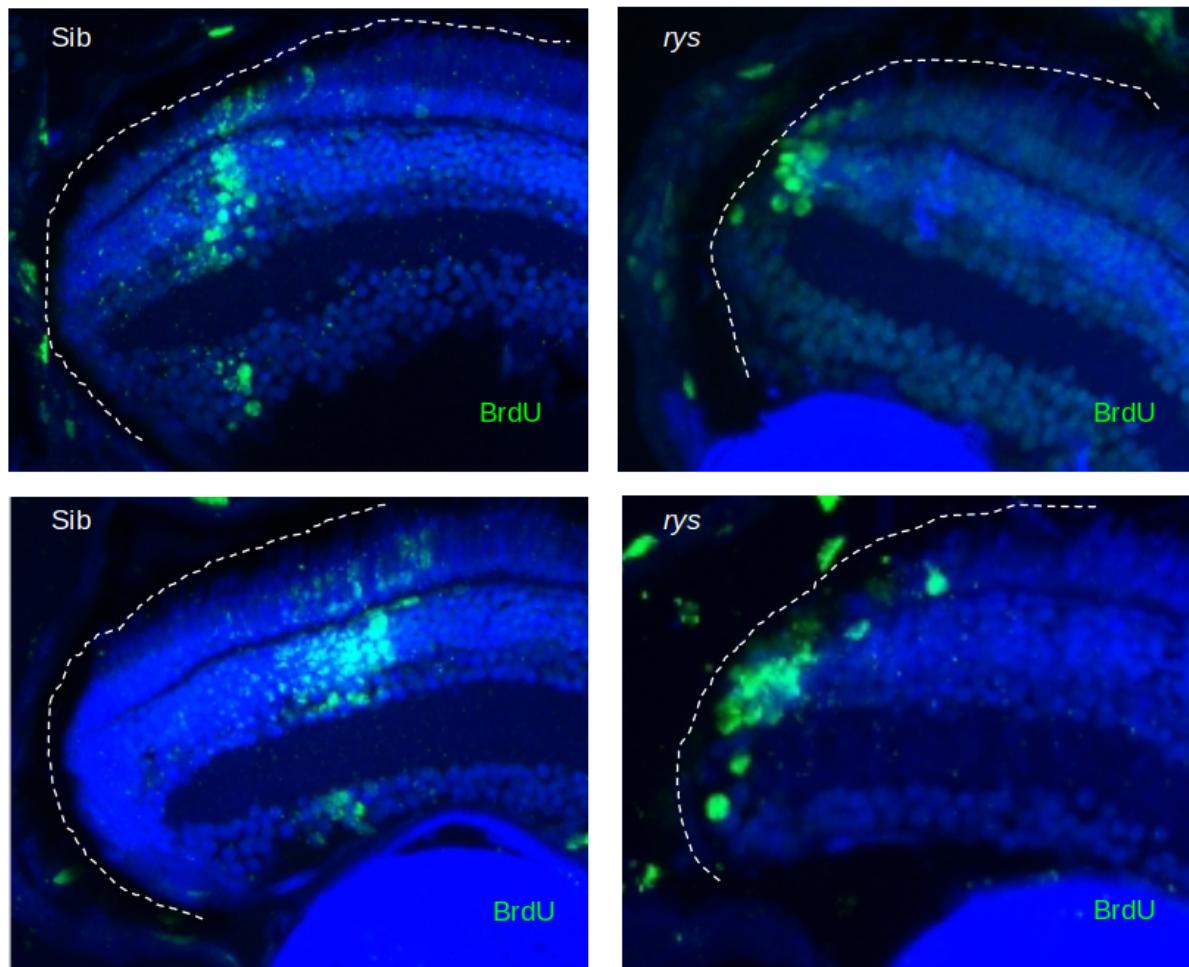


Figure 5.3: *rys* CMZ RPCs fail to contribute to the neural retina

M 14 μ m coronal cryosections through representative sib (left panels) and *rys* eyes at 10dpf, 7 days after an 8hr BrdU pulse at 3dpf. Note that few labelled *rys* cells have entered the specified retinal layers.

Indeed, a close study of the 5dpf retinae, held back from EdU processing to preserve their nuclear features (presented in Figure 5.4), suggests that the primary reason for the enlarged appearance of *rys* CMZs, even at this age, when the niche's population is numerically similar to siblings (Panel A), is this failure to contribute to the neural retina, leading to *rys* central retinae that are about half as populous, per cell of the CMZ, as their siblings (Panel B). This appearance may be enhanced in the dorsal CMZ, as 87.8% of the marginal posterior mass of the mean estimate for this region is above the sibling mean, although this is compensated for by 97.5% of the ventral marginal posterior distribution on the ventral mean laying below the same sibling mean (Panels C and D). More significantly, *rys* nuclei tend to be much larger than their siblings (Panel E), as well as consistently less spherical (Panel F).

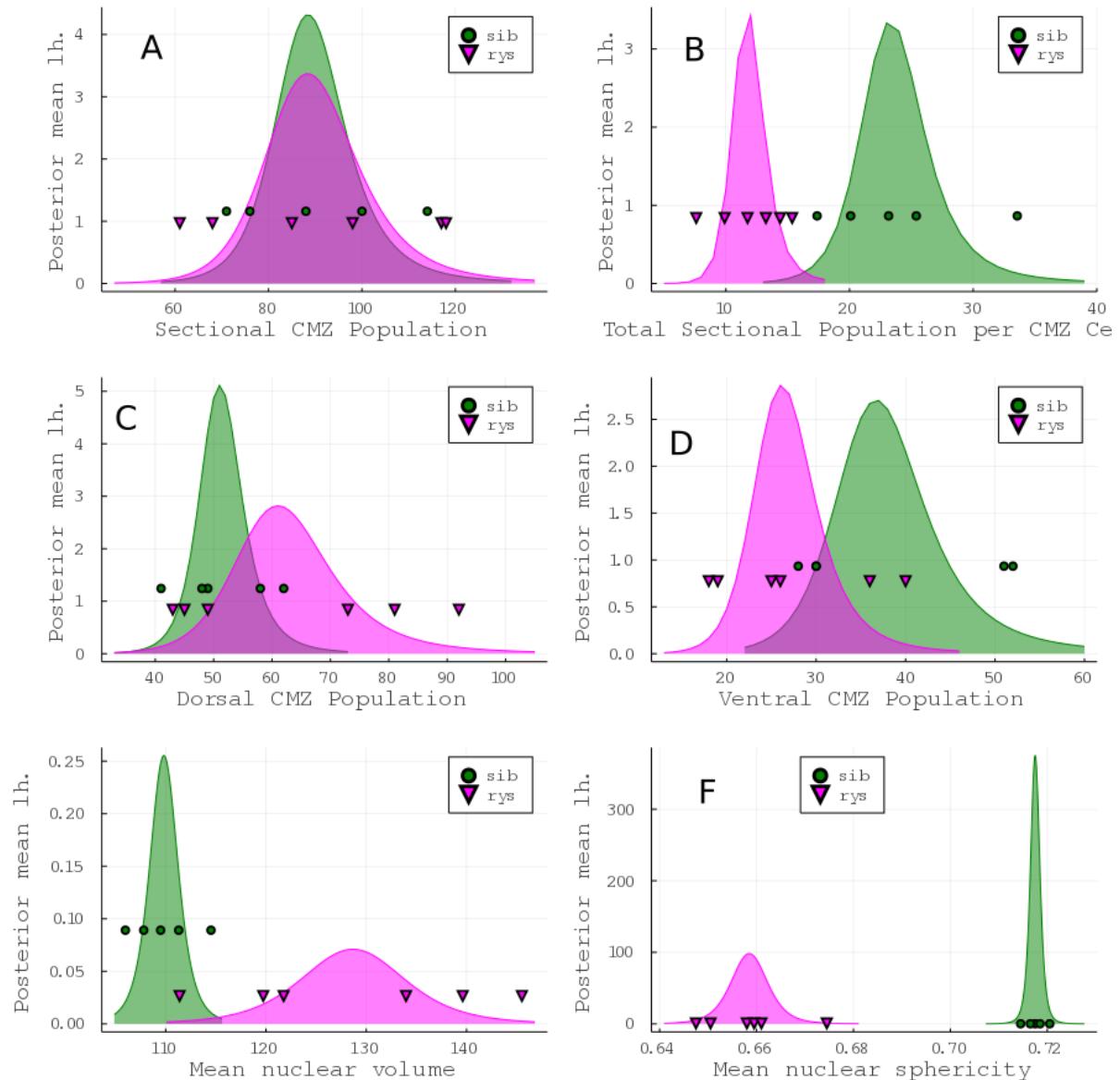


Figure 5.4: 7dpf *rys* CMZ RPCs display enhanced asymmetry, increased volume, decreased sphericity, and relative but not absolute enlargement

All panels display sib observations as green circles and *rys* as magenta triangles. Plotted behind the underlying observations are the calculated marginal posterior distributions of the mean, given log-Normal models of the population data and Normal models of the volume and sphericity data. The y-axis shows the relative likelihood of underlying mean values on the x-axis, given the data. Panel A: Total dorsal + ventral CMZ population per central coronal section. Panel B: Number of PCNA negative, specified central retinal neurons per PCNA positive CMZ cell. Panels C and D: Dorsal and Ventral CMZ populations per central coronal cryosection. Panel E: Mean volume of nuclei in a given individual's central coronal cryosection. Panel F: Mean sphericity of nuclei in a given individual's central coronal cryosection.

These characteristically enlarged nuclei in *rys* have a billowy appearance suggestive of chromosomal

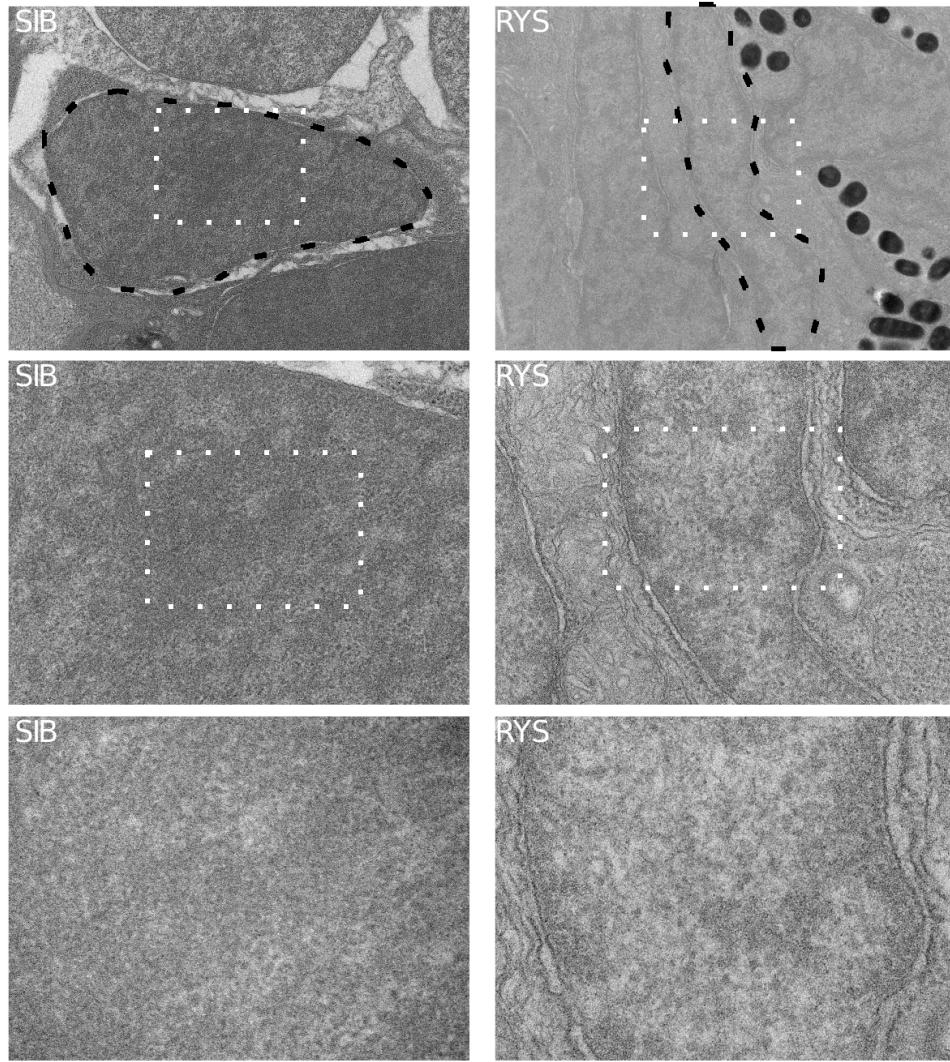


Figure 5.5: RPC nuclei of the *rys* CMZ display disorganized, loosely packed chromatin
 Representative electron micrographs of *rys* sibling and mutant CMZ RPC nuclei. Left panels: siblings. Right panels: *rys*. Top panels: 36000x magnification, general overview of the area around the nucleus, dashed black. Area displayed in middle panels dashed white. Middle panels: 110000x magnification nuclear detail. Area displayed in bottom panels dashed white. Bottom panels: 210000x magnification, chromosomal ultrastructure.

disorganization. In order to investigate this possibility further, we used electron microscopy to examine the nuclear ultrastructure of RPC nuclei in *rys* and sibling CMZs. Representative electron micrographs are presented in Figure 5.5. At 36000x magnification, the unusual and disorganized structure of *rys* nuclei become apparent, particularly in contrast with the consistently teardrop-shaped nuclei of sibling RPCs; the *rys* nucleus pictured is so pancaked it extends out of the frame that readily captures a sibling nucleus. When the chromatin itself is imaged at 210000x magnification, it appears much less electron-dense in the *rys*, with much larger tracts of presumptive euchromatin, and less regular spacing of chromosomal material.

If RPCs in *rys* CMZs are failing to enter the specified neural retina, but by 7dpf are becoming

mitotically active, this leaves the question of why this apparently proliferative niche's population is declining by 10dpf. We suspected that *ryst* RPCs may be undergoing apoptosis in situ. Although we did not detect pyknotic nuclear fragments in our EM investigations, it is possible that the individual apoptotic events are too rare in *ryst* to reliably detect in this manner. In order to investigate this possibility, we assayed the presence of caspase-3 in *ryst* and sib CMZs. As displayed in Figure 5.6, caspase-3-positive nuclei can be detected in the *ryst* CMZ but are not found in sib CMZs, though both display a similar level of central apoptotic activity. The lack of debris observable in *ryst* CMZs is likely attributable to the activity of 4C4-positive microglia active in the area; we observed one such cell actively phagocytosing an EdU-labelled *ryst* RPC in the CMZ, displayed in supplementary Figure 12.3.

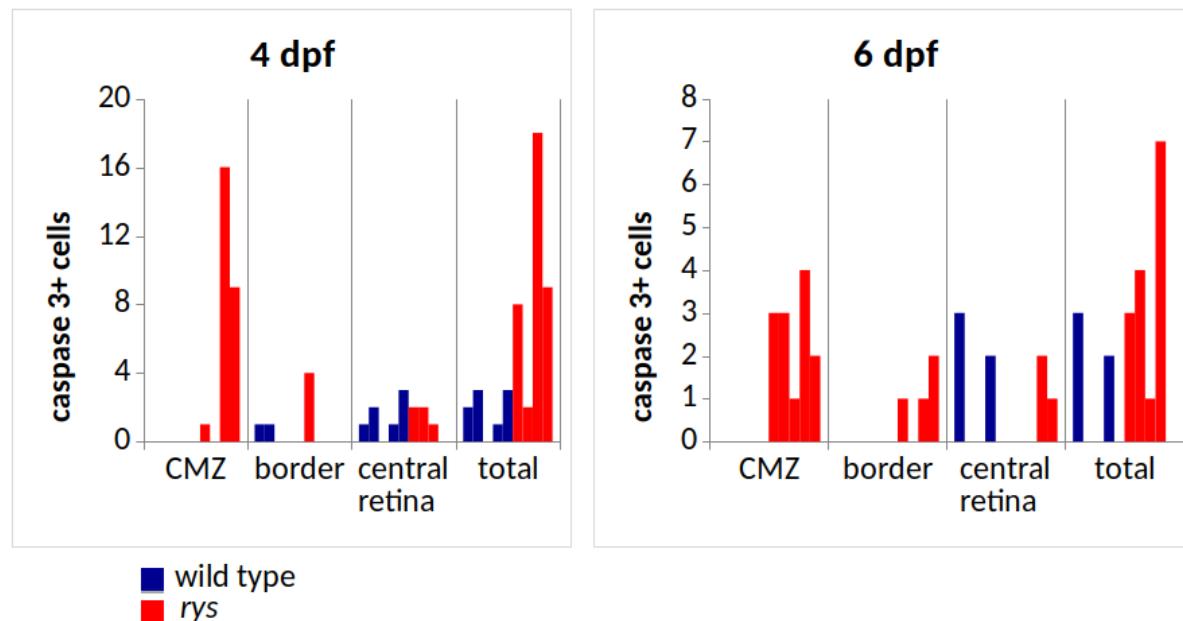


Figure 5.6: *ryst* mutant CMZs have increased caspase-3 positive nuclei

Suspecting that *ryst* CMZ RPCs may maintain an early progenitor identity, causing their failure to contribute to the specified neural retina, we examined pax6a immunostaining of this population, which is normally restricted to putative progenitors in the peripheral and middle CMZ, as well as the retinal ganglion cell layer [RBBP06]. The Pax6-stained region was enlarged in *ryst* CMZs relative to their siblings (Figure 5.7, center panels). As vsx2 is also known to be a marker of retinal progenitor cells in the CMZ [RBBP06], we also generated a transgenic vsx2::eGFP *ryst* line using a fragment of the zebrafish vsx2 promoter which drives eGFP expression in the utmost retinal periphery in siblings. Mutant fish from this line displayed substantially expanded eGFP expression in the CMZ (Figure 5.7, right panels).

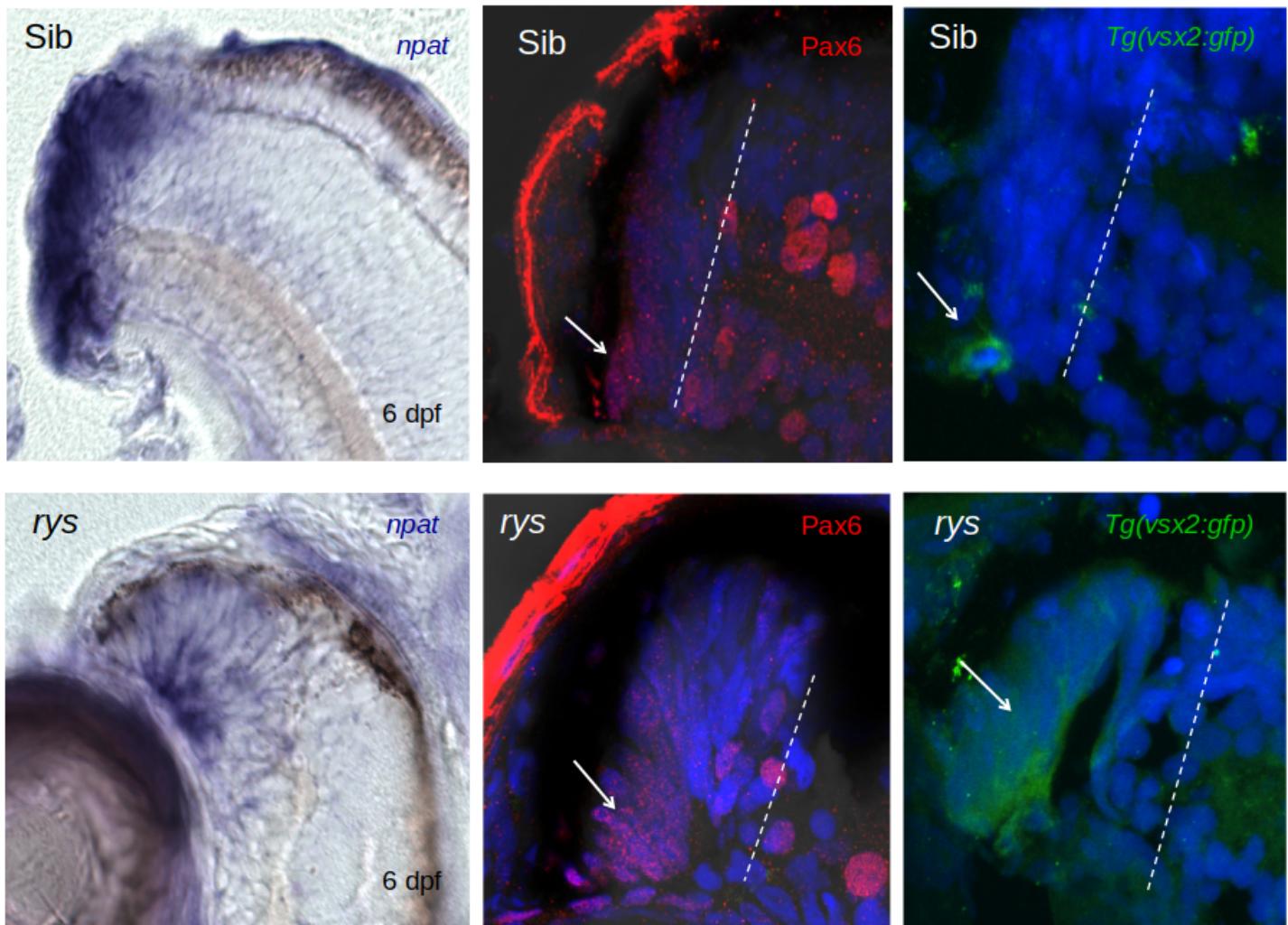


Figure 5.7: Mutant *rys* RPCs display expanded expression of early progenitor markers
 Representative transmitted light and confocal micrographs of *rys* sibling and mutant CMZ RPCs. Top panels: siblings. Bottom panels: *rys* mutants. Left panels: in-situ hybridization using npat probe on 20 μ m coronal cryosection. Middle panels: anti-Pax6 immunohistochemistry. Right panels: GFP expression in a Tg(vsx2:GFP) line introgressed into *rys*.

5.2.2 The microphthalmic zebrafish line *rys* is an npat mutant

In order to determine the causative mutation responsible for the *rys* phenotype, we performed linkage mapping to identify candidate genes, followed by PCR analysis of the transcript products of these candidates. This study revealed a single G>A transition at position 24862961 on chromosome 15 (NC_007126.5, Zv9), annotated as the first base of intron 9 in the zebrafish npat gene, in a canonical GU splice donor site.

We observed that this mutation reliably results in the retention of npat intron 8, and less frequently, in the retention of both introns 8 and 11, in 6dpf *rys* mutants, shown in Figure 5.8. The predicted protein sequence expressed from the mutant transcript is truncated by a stop codon at residue 283, which would preclude the translation of predicted phosphorylation sites and nuclear localisation signals

cognate to those identified in human NPAT [Ma00, STN⁺⁰²], as displayed in Figure 5.9.

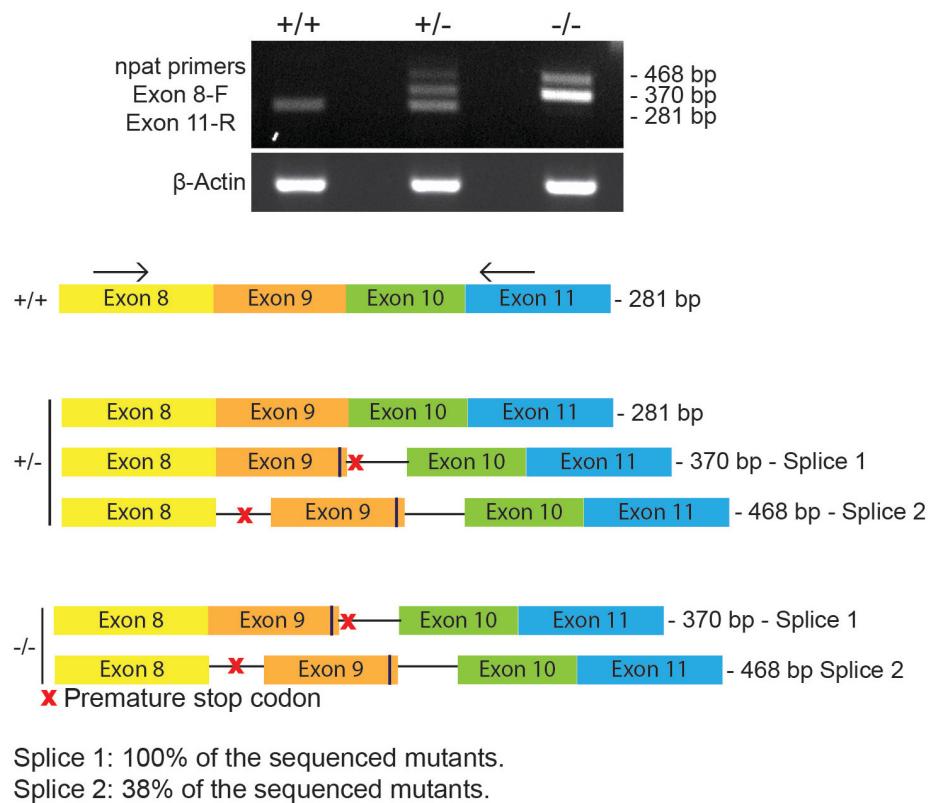


Figure 5.8: RT-PCR analysis reveals two aberrant intron retention variants in *r ys* mutant *npat* transcripts

Top panel: Agarose gel electrophoresis of mRNAs prepared from 3dpf homozygous wild type (+/+), heterozygote (+/-), and homozygous mutant (-/-) animals, as identified by genomic PCR. Fragments were amplified from a forward primer sited in exon 8 and a reverse primer sited in exon 11. Bottom panel: exon/intron layout schematics of the putative transcripts detected.

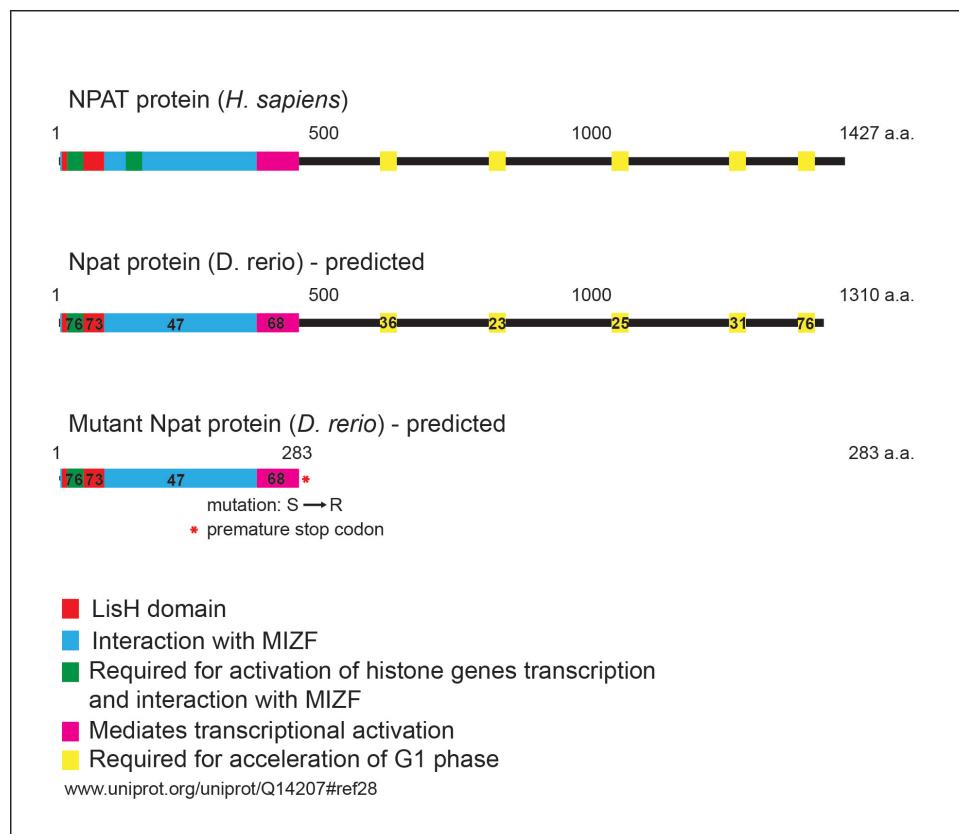


Figure 5.9: Functional domains of Human NPAT compared to predicted wild-type and *r ys* *Danio* npat

Because *D. rerio* is a teleost known to have undergone genome duplication in an ancestral clade, we used the Synteny Database tool [CCP09] identify any possible duplicates; plausibly, a duplication in zebrafish npat could explain the lessened severity of the mutant phenotype when compared to mammalian proviral inactivators. The results of this analysis are presented in supplementary Figure 12.1. While npat is in the midst of a region which appears to be duplicated on chromosomes 5 and 15, relative to the unduplicated homologous synteny run on *H. sapiens* chromosome 11, it is not, itself, duplicated.

We performed in situ hybridisation using a probe directed to the wild type npat transcript to confirm that the gene is indeed expressed in wild type CMZs; we found that npat expression is progressively restricted to the CMZ from 4 to 6 dpf in wild-type fish. A representative time-course of 20 µm cryosections through ISH-treated embryos is depicted in Figure 5.10, focusing on the retina at the times when it has formed. While npat remains transcribed in both the specified GCL and amacrine-rich inner INL to a degree, it is most intensely expressed in the proliferative CMZ, as we might expect on the basis of its cell cycle functions.

Having taken note of the unusual chromatin ultrastructure present in *r ys* CMZ RPCs, and with a mutant npat allele reliably linked to the appearance of the *r ys* phenotype, we investigated the transcriptional status of npat and its histone regulatory targets in these animals. We first assayed npat itself by RT-PCR, finding that homozygous *r ys* mutants overtranscribe npat by about 3-fold compared to their wild-type counterparts at both 6dpf and 8dpf, while sibling overabundance declines from about 2.5-fold

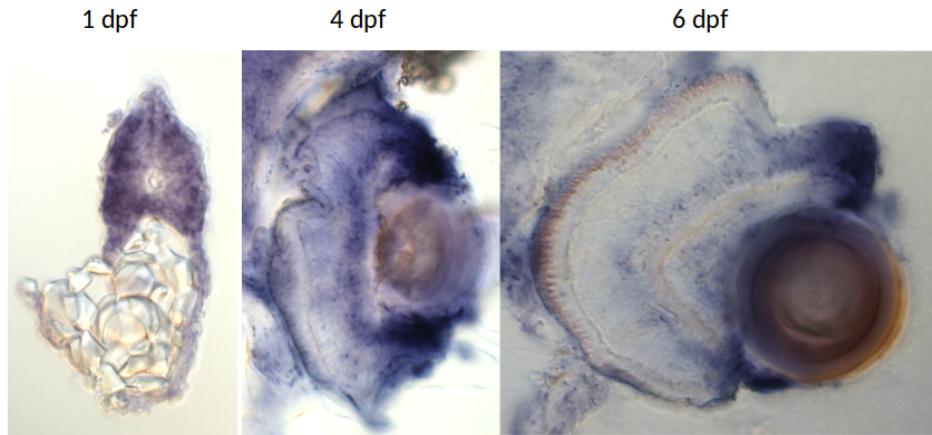


Figure 5.10: **In situ hybridization reveals progressive restriction of npat expression to the CMZ**

20 μ m sections of wild-type embryo (1dpf) and retinae (4 and 6 dpf), displaying progressive restriction of npat expression as assayed by in-situ hybridisation.

to 1.5-fold over this time period, as shown in ??, with calculated marginal posterior mean mass over the WT standard given in ??.

Since mammalian NPAT is known to regulate histone transcription and is critical for coordinating the correct expression of replication-dependent histone transcripts required to package genomic DNA during S-phase [ZKL⁺00], we hypothesized that the altered nuclear morphology and truncated S-phase we observed in proliferating *rzs* CMZ cells may be a consequence of perturbed regulation of histone transcription. To test this, we performed qPCR on random-hexamer-primed cDNAs produced from 6 and 8dpf wild type, sibling, and *rzs* embryo mRNA extracts. These qPCR assays were performed using degenerate primers² directed toward all members of the zebrafish core histone gene families H2A, H2B, H3 and H4. As a role for NPAT in 3' end processing of histone transcripts has been identified [PJ10], we also set out to determine whether 3' end processing of histone transcripts is altered in *rzs*. By repeating the above-described qPCR assays on oligo-dT-primed cDNAs, we were able to examine the population of polyadenylated histone transcripts in isolation.

In 6dpf fish, we observed increases in the levels of H2A, H2B, and H3 family transcripts in *rzs* cDNA compared to wild type (Figure 5.11, Panel A). Transcripts for core histone proteins H2A, H2B, and H3 were notably overexpressed by 6dpf in mutants, and by 8dpf all of H2A, H2B, H3, and H4 are notably overexpressed by approximately 2 to 6-fold in both siblings and *rzs*, relative to wild-type. Calculated marginal posterior mean mass over the WT standard, and, for *rzs*, the sibling mean, are given in ?. Even more pronounced in *rzs* mutants, but not siblings, were the overabundances of polyadenylated core histone transcripts, particularly in H2B but noted in all core histone classes at one or both times. Calculated marginal posterior mass of the mean above the WT standard value are displayed in ??.

While these observations substantiate the involvement of npat in the *rzs* phenotype, we pursued the matter further by perturbing npat using morpholino injections of wild-type fish. As methodological issues prevented us from identifying the specific npat species expressed in *rzs* (whether abberantly

²Zebrafish have notably populous histone clusters with numerous variants and pseudogenes not present in non-genomically-duplicated vertebrates, so degenerate primers are an appropriate way to survey the population of transcripts. A full catalogue has not been undertaken, to my knowledge.

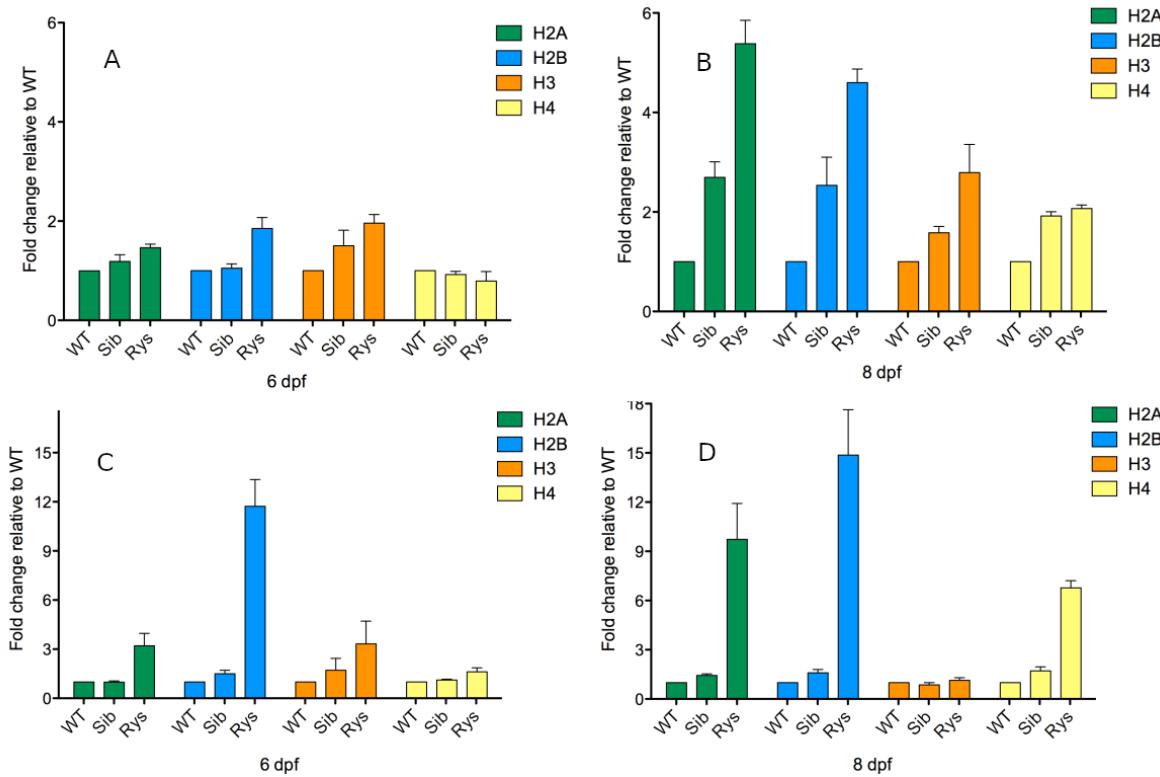


Figure 5.11: *ryst* overexpress total and polyadenylated core histone transcripts
qPCR results for degenerately-primed families of core histone transcripts, from random hexamer primers for panels A and B, measuring total core histone transcripts, and oligo-dT for panels C and D, measuring polyadenylated transcripts. Panels A and C display results from cDNAs of 6dpf *ryst* mutants and siblings compared to wild-type conspecifics, while B and D are from 8dpf animals.

truncated or otherwise), we sought not to recapitulate the *ryst* phenotype but to determine if any of its constituent phenomena would appear after an early blockage of *ryst* transcription, substantiating the general involvement of npat in *ryst*. We tested morpholinos directed both to the npat start codon and to the splice site affected in *ryst*, alongside control morpholinos and uninjected animals. We used both a morpholino directed to the transcript ATG start site, as well as to the affected splice site. The splice morpholino consistently replicated the *ryst* phenotype on both total and polyadenylated core histone transcript abundance, while the ATG morpholino more narrowly replicated the effect on polyadenylated transcript (??). The ATG morpholino produced animals with small eyes by 72dpf, as shown in Figure 5.12, and confirmed by census of the cellular population of central coronal sections through the retina ???. These results establish that the identification of npat as the causative mutation in *ryst* is plausible, as experimental perturbation to npat in WT fish can cause *ryst* phenotypic phenomena.

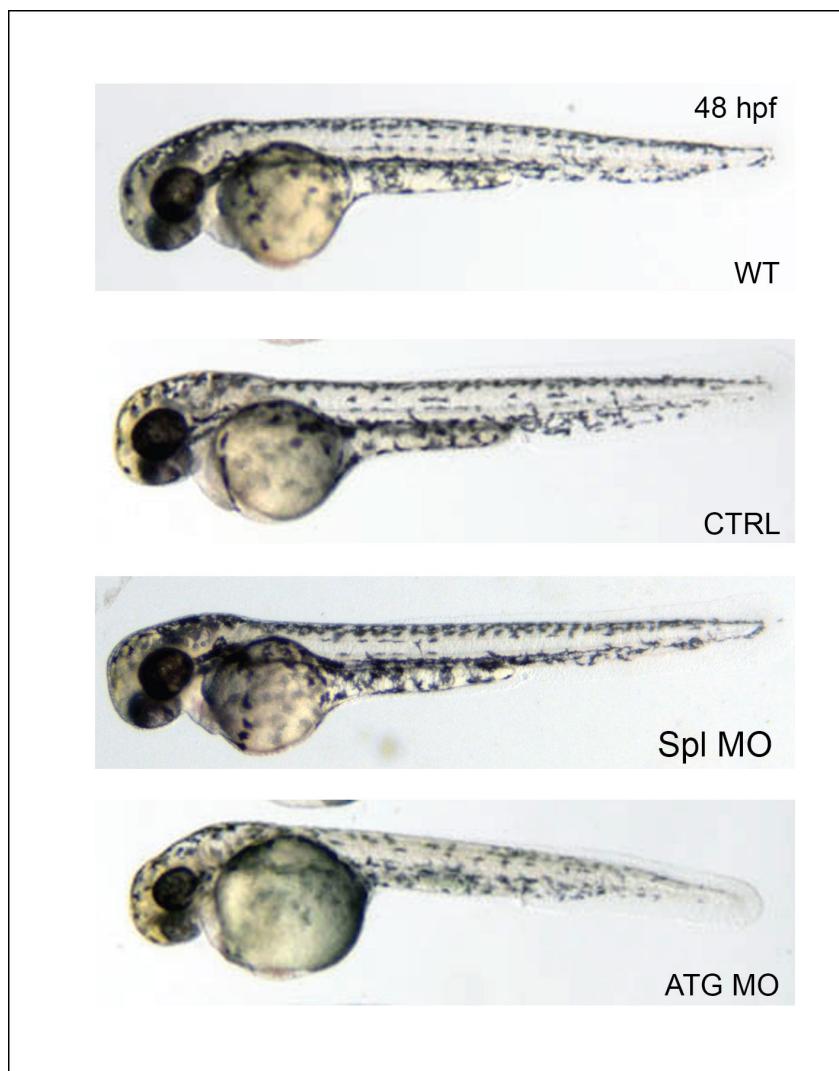


Figure 5.12: 48hpf embryos injected with an npat ATG-targeted morpholino display a small-eye phenotype

20 μ m sections of wild-type embryo (1dpf) and retinae (4 and 6 dpf), displaying progressive restriction of npat expression as assayed by in-situ hybridisation.

5.2.3 *rys* siblings and mutants have unique sets of nucleosome positions, best explained by different sequence preferences and increased sequence-dependent positioning in mutants

Our observation of perturbed histone expression in *rys* embryos led us to suspect that the aberrant nuclear morphology observed in *rys* CMZ progenitors arises from disrupted chromatin organisation, which is a possible consequence of altering the dynamic composition of the histone pool available for nucleosome formation during cell cycle. We therefore sought to characterise nucleosome positioning in *rys* and wild-type siblings by micrococcal nuclease (MNase) digestion of pooled genomic DNA (gDNA). Nucleosome-protected fragments from the MNase digests were sequenced and positions called as described previously.

We first examined the genomic disposition of nucleosome positions within wild-type siblings and *rys*. We found nucleosome positions distributed relatively evenly over sib genomic material, with only tiny deviations from a neutral assumption of a uniform distribution of the total position number over the full length of the genome, as displayed in Fig. 5.13, panel A. Bulk *rys* chromatin displays minor differences from the proportions of positions found in each sib scaffold (panel B). Sib nucleosome positions are differentially occupied across scaffolds, with Chr 10 and scaffolds not yet mapped to chromosomes (NC) being notably more occupied than expected from scaffold length alone (panel C). Similarly to the distribution of positions, *rys* chromatin is somewhat more evenly occupied than sibs; scaffolds with positions that are more heavily occupied in sibs tend to be depleted in *rys* and vice versa (panel D).

Mindful of the whole-organism source of the nucleosomal genomic sample used to generate the called positions, and of the heterogenous nature of the *rys* phenotype, with RPCs displaying the nuclear phenotype but not specified retinal neurons, we sought to identify position subpopulations that might represent those involved in the nuclear phenotype. By mapping the called nucleosome positions in *rys* to those in sibs, we observed that there is a subset of sib positions which are never found to be occupied in *rys*, and likewise a subset of *rys* positions which are unique to the mutants (Fig 5.14). Intriguingly, sib positions which are not observed in *rys* are compensated for quite evenly across the genome by new positions gained in the mutants, with a small excess of new *rys* positions on most chromosomes.

This result suggested that a particular subpopulation of wild type nucleosomes are mislocalised in *rys*. If the pool of available histones in proliferating *rys* progenitors is substantially different from that of siblings, *rys* nucleosomes forming from this pool may have altered physico-chemical interactions with DNA, which could result in a preponderence of nucleosomes with unusual sequence preferences. If so, this would explain the disappearance of a subset of positions in *rys* and the appearance of a new, similarly sized subset of positions (the ‘differential set’).

To formally address this hypothesis, we calculated the Bayesian evidence ratio for separate emission processes for sib and *rys* position sequences against a combined process for both sets of sequences. If the molecular process resulting in the differential set arises from altered nucleosome composition in proliferating *rys* cells, we expect these sequences to provide greater support for separate emission processes than for a single, combined process. On the other hand, if aberrant DNA-nucleosome interaction is not causally relevant, and the reason for the apparently displaced nucleosomes is another macromolecular process, the evidence ratio should favour a combined process for the emission of the differential set—that is, there should be no difference between the unique sib and *rys* sequences that would justify the additional complexity of separate models.

The emission model used was the Independent Component Analysis form described by Down and Hubbard [DH05], with a fixed number of independent, variable length position weight matrices related to observations by a Boolean mixing matrix. An observation is scored by the background likelihood of its sequence, given some model of genomic noise, convolved with a sequence-length- and cardinality-penalized score³ for each source which the mix matrix indicates is present in the observation. Therefore, we first needed to construct background models of *D. rerio*’s genomic “noise” from which repetitive sequence signals characteristic of nucleosome positions could be extracted. Following the suggestion of Down and Hubbard [DH05] that a principled approach to the selection of background models is to train and test a variety of them on relevant sequence, we used the Julia package BioBackgroundModels

³That is, the additional score provided by a source to an observation is penalized both by the expected number of motif occurrences given an observation of that length, as well as by the number of sources which explain the observation in the model.

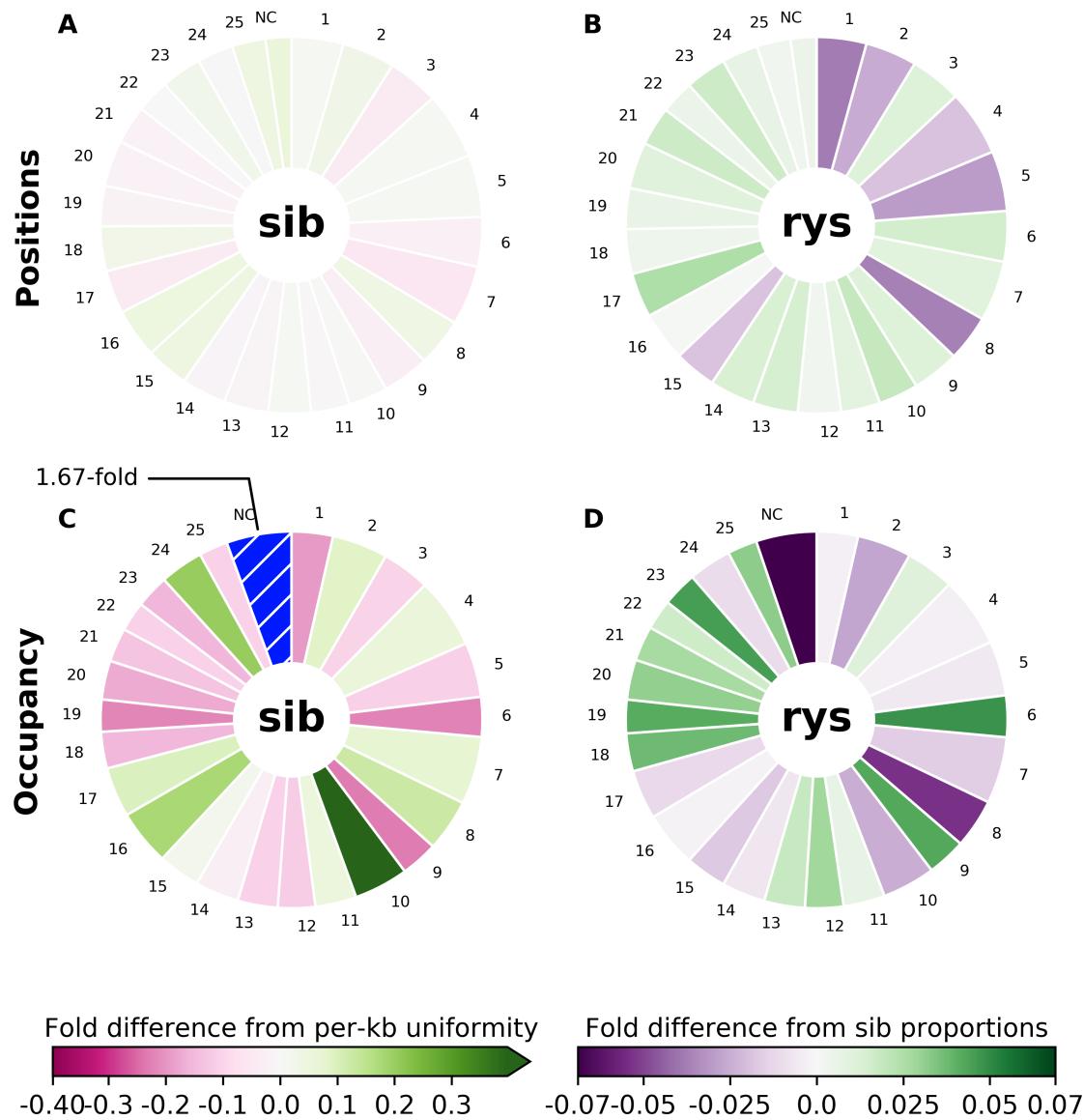


Figure 5.13: *rys* chromosomes are differentially enriched and depleted of nucleosome position density and occupancy.

Pie charts of nucleosome position density and occupancy by chromosome. Width of pie slices in panels A and B indicates the fraction of the total number of positions occurring in the numbered chromosomes and nonchromosomal scaffolds (NC). In panel A, depicting the *sib* genome, slices are colored according to the extent of deviation from an assumption of even distribution of nucleosomes across the genome. In panel B, depicting the *rys* genome, slices are colored according to the extent of deviation from the *sib* distribution. The width of slices in slices C and D indicate the fraction of the total nucleosome occupancy signal detected. Slice coloration depicts deviations from nucleosome occupancy distributions analogous to position distributions in A and B. Blue and white diagonal bars in the NC slice of panel C denote an out-of-scale positive deviation from the assumption of even distribution, i.e., *sib* NC scaffolds have 1.67-fold more of the total nucleosome occupancy signal than is expected from their length alone.

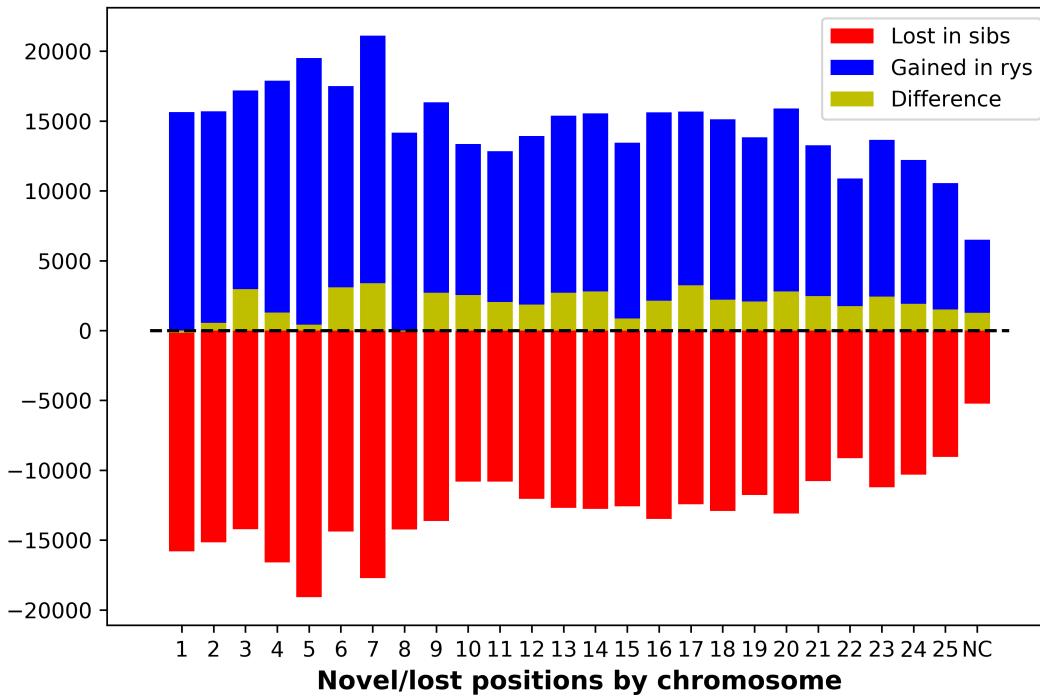


Figure 5.14: **Novel nucleosome positions in *rys* occur in similar numbers to those lost from sibs.**

Counts of positions found in sib but not *rys* (red bars, represented as negative numbers, as these are ‘lost’ in *rys*) and those found in *rys* but not sib (blue bars, ‘gained’). The magnitude of the difference between the counts is represented with a yellow bar.

(presented in Chapter 8) to screen a panel of 1,2,4, and 6-state HMMs against 0th, 1st, and 2nd order encodings of samples from the zebrafish genome, partitioned grossly into exonic, periexonic, and intergenic sequences. We found that each of these partitions is best represented by 6-state HMMs trained on a 0th order genome encoding (i.e. the HMMs emit the 4 mononucleotides), as determined by model likelihood given an independent test sample, as displayed in ??.

We next used the Julia independent PWM component analysis nested sampling library BioMotifInference (presented in Chapter 9) to sample from the posterior distribution, given the differential set of *rys* nucleosome positions and the composite background model of genomic noise. We initialized separate ensembles from uninformative priors on the *rys* and sib data alone, as well as the combined *rys* and sib data, allowing for 8 independent PWM sources in all cases. We compressed three model ensembles to within 125 orders of magnitude between the maximum likelihood model sampled and the minimum ensemble likelihood⁴. This process produced the model evidence estimates summarized in ??, as well as maximum a posteriori samples for each of the ensembles. We estimate that there are orders of magnitude of evidence in favour of separate generative processes for the differential sib and *rys* than for a combined model, with an estimated standard deviations of significance. We present the MAP PWM source samples from the better-evidentiated separate *sib* and *rys* models in Figure 5.15 and Figure 5.16

⁴That is, the convergence criterion was that the ensemble difference $\log(L_{max}) - \log(L_{min})$ be <125).

respectively, while the inferred combined sources are available in supplementary Figure 12.4.

BioMotifInference's source detection was highly conservative, likely due to the good quality of the background models, which were found to explain a majority of observed nucleosome positions adequately without any PWM sources in most models of the maximum a posteriori estimate for both sibling and rys ensembles. In siblings, the top three sources are the only ones detected in more than 10% of observed sequences, suggesting that the influence of sequence preferences is weakly explanatory for these sites. By contrast, we found twenty-eight separate PWM sources expressed in more than 10% of observations, in a variety of posterior modes. This suggests sequence preferences are much more explanatory for the *rys* differential position set.

The detected sources in the siblings are not altogether unexpected; CWG motifs are among the most commonly reported in nucleosome sequences, and have been described as promoting nucleosome formation. The majority of the sources detected in the various posterior modes were of this form, with rarer CT- and CA- dinucleotide repeats, as well as an rare ATGG repeat. *rys* positions have a much more diverse set of sources detectable above background genomic noise; notably, AG- dinucleotide repeats that are not found in *sib* sources at all. The CWG motifs also exhibit a preference for flanking A positions that is not evident in the sibling differential position set. CA- dinucleotide repeats are more commonly detected in *rys* positions, and the rare ATGG repeat is not found at all.

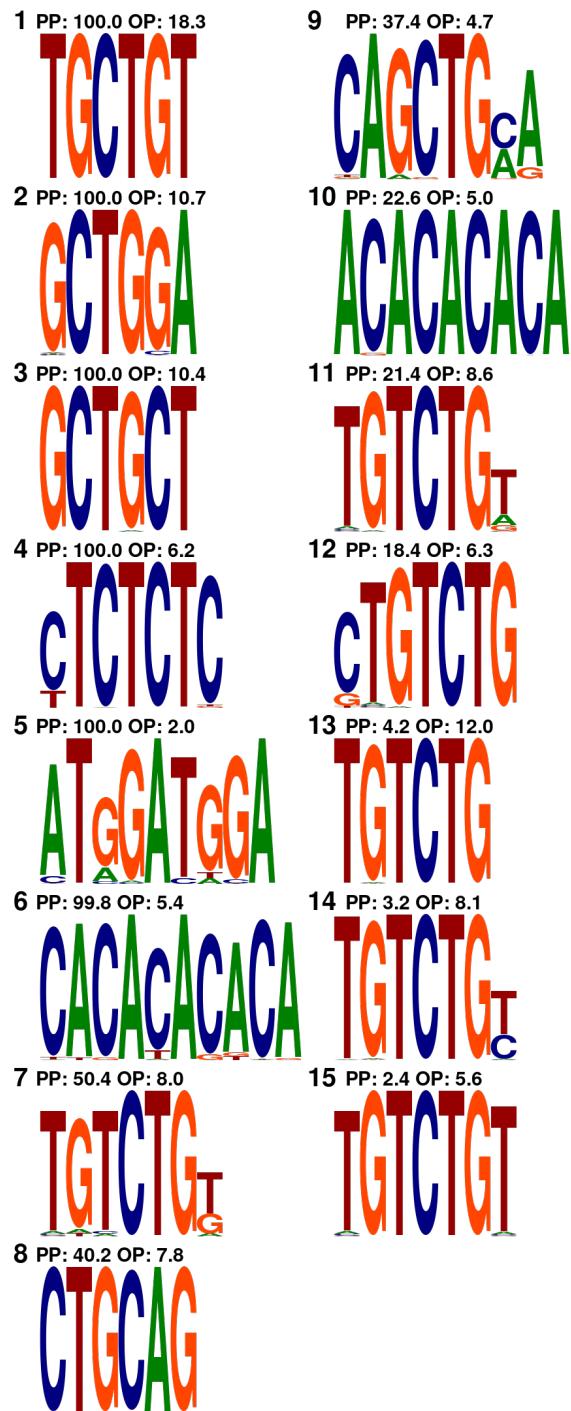
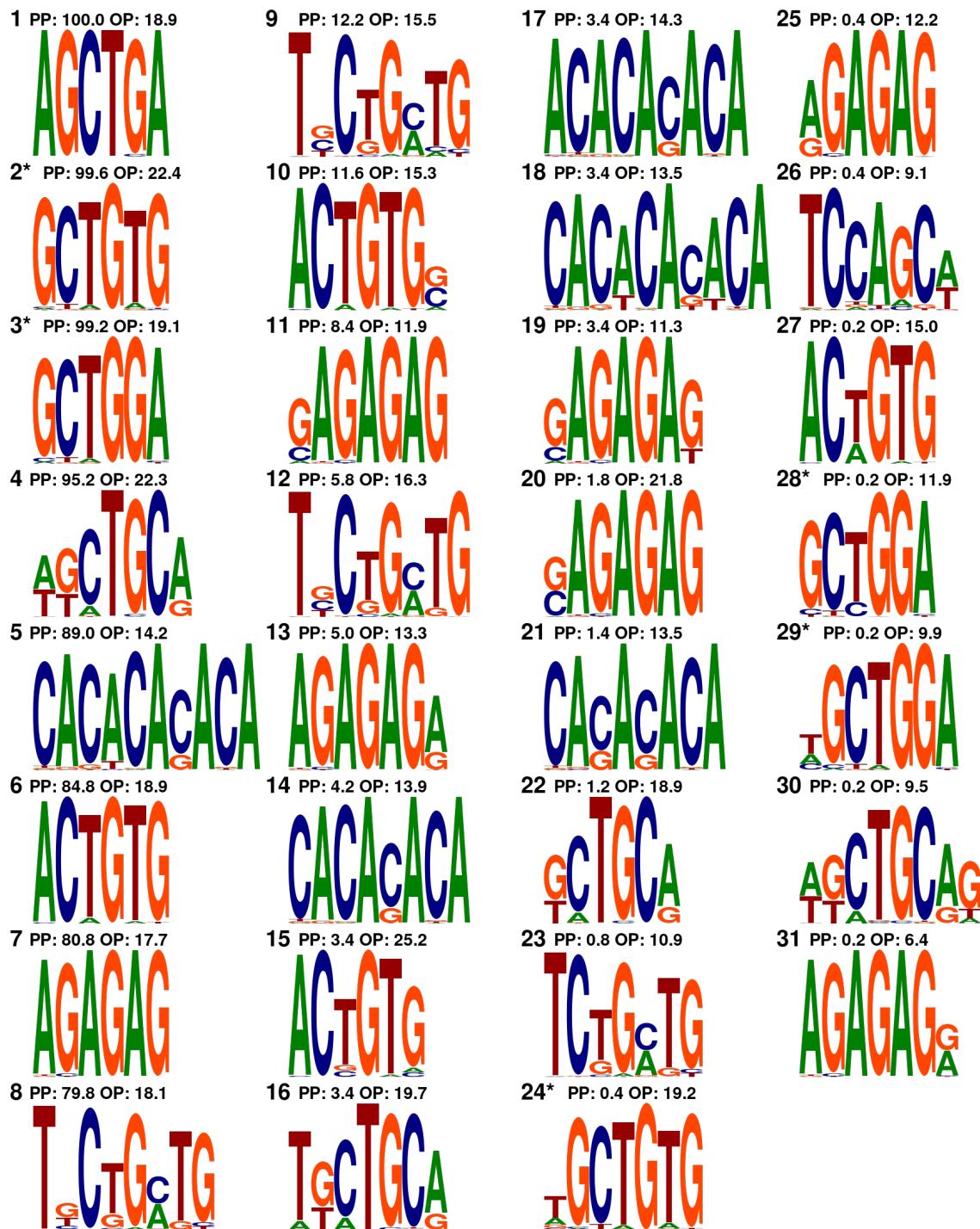


Figure 5.15: PWM sources detected in sibling differential nucleosome positions.
Counts of positions found in sib but not *rys* (red bars, represented as negative numbers, as these are ‘lost’ in *rys*) and those found in *rys* but not sib (blue bars, ‘gained’). The magnitude of the difference between the counts is represented with a yellow bar.

Figure 5.16: PWM sources detected in *rys* mutant differential sources.

Counts of positions found in sib but not *rys* (red bars, represented as negative numbers, as these are ‘lost’ in *rys*) and those found in *rys* but not sib (blue bars, ‘gained’). The magnitude of the difference between the counts is represented with a yellow bar.

5.3 Discussion

We began our investigations by adding depth to the original description of the *rjs* mutant CMZ phenotype; on the basis of these investigations, we believe that a revision is in order. At any given time, the apparently increased size of the CMZ in *rjs*, relative to siblings, is produced as an effect of one or more of the following phenomena:

1. Inappropriately retained RPCs in animals older than 7dpf
2. Reduced neural population of the specified central retina, relative to the CMZ
3. Expanded and spread-out RPC nuclei

The first two effects are consistent with a failure of RPCs to specify as particular retinal neural lineages. This is substantiated by the inappropriate retention of early progenitor markers in these cells. We demonstrate that *rjs* RPCs are not, however, arrested in cell cycle. In contradistinction to results finding polyadenylated histone transcript associated with cell cycle arrest [KKT⁺13], we find that the mitotic activity of these cells actually accelerates as the overabundance of these transcripts increases. We therefore suggest that the *rjs* phenotype is characterised by chromatin disorganisation (phenomenon 3) and a failure of RPCs to correctly specify (which covers phenomena 1 and 2) as retinal neural subtypes. The enlarged appearance of the CMZ is a result of these effects, and not a general increase in the CMZ population, RPC cytoplasmic mass, or other variables.

Moreover, in identifying npat as the lesioned gene underlying the *rjs* phenotype, we provide a general, well-evidenced macromolecular mechanism underlying phenomenon 3 which could plausibly produce phenomena 1 and 2. That is, the mutant npat's effects on cell-cycle dependent histone transcription and stability may result in aberrant nucleosome positioning within proliferating cells by altering the pool of histone proteins available for nucleosome formation. We posit that this is what causes the observed loss of a subpopulation of sibling nucleosome positions in *rjs*, together with the gain of novel, aberrant positions. When we investigated whether the sequences of these subpopulations were better modelled by separate emissions processes than a combined emissions process, we found substantial evidence in favour of separate emissions processes. Interestingly, the PWM signals we detected in this differential nucleosome position set suggest some detail as to how changes in histone expression might result in the observed nuclear disorganisation of *rjs*. The observation that the background models explain the positions unique to siblings better than those unique to mutant *rjs* strongly suggests that sequence preference is less important in the process generating sibling positions than in the one generating mutant positions.

The relative importance of primary sequence in nucleosome positioning has been hotly debated; some have advocated for a nucleosome positioning code intrinsic to primary sequence [KMF⁺09], while others have disavowed the existence of any such code [ZMR⁺09]. In general, the nucleosome dynamics community has moved on from these discussions in favour of emphasis on rotational sequence preferences [TCM⁺07] and translational nucleosome positioning by *trans*-acting factors [KKZ⁺18]; a common interpretation of the earlier debate is that sequence preferences tend to dominate only at limiting concentrations of histones, when chromatin formation is rare [Poi13]. The fact that sequence preference is less explanatory for the sibling positions than for the mutant ones is highly suggestive against this background. It seems likely that the shifted and novel nucleosome positions observed in mutants reflects all of the following:

1. The appearance of nucleosomes with aberrant subunit composition
2. A loss of translational control over *rys* mutant nucleosomes by *trans*-acting factors
3. The limiting concentration of aberrantly-expressed nucleosomes, resulting in increased influence of “default” sequence-preference positioning

All of the above could plausibly be caused by perturbed histone expression arising from altered npat activity. As the spatiotemporal dynamics of chromatin architecture are known to be critically involved in both replication timing [GTR⁺10] and cell identity and fate [SVT13], the significant alterations we observe in *rys* mutant chromatin architecture also provide a plausible molecular mechanism by which mutated npat protein can produce the observed shift in proliferative activity and failure of RPCs to specify, via altered histone expression. Moreover, chromatin density has recently been specifically implicated as fundamentally involved in the process by which pluripotent cells restrict gene expression to achieve their stable specified fates [GJH⁺17]; it seems to be this overall process which has been disrupted in mutant *rys*.

It is possible that the sequence-level inference is compromised by the ad-hoc nature of the sampler used to compress the ICA ensembles in our nested sampling procedure. For reasons explained in Chapter 9, the ICA model structure makes euclidean space representations difficult, and ensuring sampling is in detailed balance may not be possible, given changing PWM signal lengths. We suggest that it is appropriate to magnify our error estimate by several of magnitude as a consequence. If we concede that there are perhaps 5 orders of magnitude more error than estimated, this has the effect of reducing the significance of the result from to standard deviations, which is still well above the typical 5σ significance typically accepted as a discovery in the physical sciences. We conclude that even given a more conservative estimate of the sampling processes’ error, it is far more likely that the differential nucleosome populations found in *rys* mutants and their phenotypically normal siblings are the result of separate generative processes. It is also possible that our identification of the differential subset of nucleosome positions with the disordered chromatin present in *rys* RPCs is inaccurate. We have not conducted a full survey of *rys* proliferative niches, and it is not clear how widely affected other cell types might be. Still, within the retina, only proliferative cells seem to have the chromatin phenotype, and it is on this basis that we make the identification, which is the most parsimonious available.

There are a number of important lacunae in this explanation; it remains unclear what form of the npat protein is expressed, if any, as well as what the overall effect on the pool of histones in RPCs is, for instance. It would not be very surprising if there are a number of macromolecular species intimately involved in the *rys* pathology which intermediate between the npat lesion and the observed chromatin phenotype which we have not examined. We suggest, however, that the overall effect on specification must be due to a failure of RPCs to organize chromatin for specification, and that, interestingly, this failure does not prevent proliferation. *rys* therefore presents a useful model of the dissociability of proliferative and specific behaviours in neural progenitors, which has recently been documented elsewhere in the developing *D. rerio* retina [ESY⁺17]. Given the methodological limitations we encountered in probing protein expression in *rys* RPCs, we suggest the most interesting and productive avenues of research to pursue in *rys* pertain to this chromatin organisation phenotype. Further experimentation is required to determine how specific changes in chromosome organisation (e.g. alterations in 3-dimensional chromatin organisation of gene expression, number of replication foci, etc.) produce specific features of the *rys* phenotype.

The zebrafish *rys* mutant model provides a heretofore unique opportunity to study the role of a human NPAT homologue in a complete, developing tissue. This has previously proven difficult using mouse Npat, proviral inactivation of which resulted in early embryonic arrest, prior to tissue formation (Di Frusco 1997). The survival and development of *rys* mutant embryos beyond this stage may reflect the presence of wild-type npat transcript contributed maternally [HSK⁺13]; *rys* mutants nevertheless do not survive beyond metapmorphosis (~21dpf), so npat seems to be similarly obligatory for normal development in zebrafish, if over a longer timeframe. Still, we observed many differences between the function of npat in zebrafish and documented effects in other vertebrates, which require some explication.

As teleost fish are known to have undergone whole-genome duplication subsequent to their radiation from vertebrates, it is possible that zebrafish may have multiple npat paralogues. We have excluded this possibility due to our failure to identify any significantly similar CDS sequences in the Zv9 zebrafish genome using BLAST, and the synteny analysis in Figure 12.1. The zebrafish npat gene does have a substantially different genomic context from human NPAT, however: it is not associated with the eponymous ATM locus (which has been duplicated, and is present in this duplicate form elsewhere on chromosome 15). ATM's 5' position is, in zebrafish, occupied by hif1al, with the intergenic region lacking canonical E2F promoter sites. Of the 80 vertebrate genomes currently available from Ensembl, this organisation is shared only with the cave fish (*A. mexicanus*), with the human-like ATM-NPAT association preserved in all other species. This apparently evolutionarily novel genomic organisation for npat may be responsible for some of the differences in npat function we report here, relative to its homologues.

We identified alterations in histone mRNA transcription and 3' end processing as likely mechanistically involved in the *rys* phenotype. In both 6 and 8 dpf *rys* larvae as well as npat-morpholino treated 1dpf embryos, we observe increased abundances of histone and, specifically, polyadenylated histone transcripts, although the composition of affected histone families differs between these contexts. The specific mechanism by which these effects are produced in *rys* remains unclear, and the increases in histone transcript abundance observed in both mutant and morpholino-treated animals is at odds with observations that NPAT knockouts display decreases in histone transcription [YWNH03], and that the destruction of CDK phosphorylation sites on NPAT protein, which would be the case in the putatively truncated *rys* npat protein, or treatment with a CDK inhibitor, result in similar declines in histone transcript abundance [Ma00, MGv⁺09]. This may imply the role of zebrafish npat in regulating histone transcription is different from what has been described for human NPAT. We are unable to determine from these data whether the overall increases histone transcript abundance are a consequence of increased histone transcription or of the greater stability of improperly polyadenylated histone transcript, however. The increased abundance of polyadenylated transcript in *rys* mutants and npat morpholino-treated embryos is consistent with the observed role of NPAT in recruiting CDK9 to replication-dependent histone gene clusters, known to be important in generating the normal stem-loop structure at the 3' end of these transcripts [PSS⁺09], suggesting that this role is conserved in zebrafish. It is also possible that particular histone genes give rise to polyadenylated transcripts in zebrafish, as observed in a variety of cell lines [KKT⁺13]; increased transcription of these particular genes might also account for the observed increase in polyadenylated histone transcript abundance. As noted above, although increased abundance of polyadenylated histone transcript has been associated with both cell cycle arrest and differentiation [KKT⁺13], we found that in the *rys* mutant CMZ, this was associated with altered cell cycle parameters, failure to differentiate, and ultimately, cell death by apoptosis. This suggests that the presence of

Polyadenylated histone transcripts are not directly related to particular cell cycle states or differentiated fates, but rather that a particular regime of coordination and control of the expression of polyadenylated and replication-dependent, stem-looped histone transcripts is required to achieve appropriate transitions between these states. In the case of *rys*, this seems to be related to the effect of altered histone expression on chromatin structure.

Chapter 6

Inferring and modelling nuclear dynamics in retinal progenitors

6.1 Intro

transition from discussion of modelling postembryonic CMZ and using nuclear shape to infer identity for modelling purposes to discussion of the implication of the correlations of nuclear state, cellular identity, and cellular function

6.2 Frontiers of nuclear dynamics

discuss recent experiments measuring chromatin conformation, pulsatile transcription etc

6.3 Integrating nuclear dynamics into models of CMZ- abstraction and biosemiotics

abstraction necessary due to computational limits, biosemiotics allow for the calibration of this theoretically

Part II

Software Technical Reports

Chapter 7

GMC_NS.jl

`GMC_NS.jl` implements a basic Galilean Monte Carlo nested sampler [Ski12, Ski19] in pure Julia. It uses the updated Galilean reflection scheme of Skilling’s later GMC publication [Ski19], which improves the algorithm’s performance and preserves detailed balance. It uses the natural attrition of model-particles getting stuck in posterior modes to regulate the number of live particles in the model ensemble. This approach achieves the same end as algorithms which dynamically adjust the ensemble size based on parameters of the ensemble [FHB09, HHHL19], without any computational overhead or need for tuning on the part of the user. A minimum ensemble size may be supplied by the user, which is maintained by initializing a new trajectory isotropically from the position of an existing particle; this ensures that the ensemble may be sampled to convergence even if the number of trajectories started with is insufficient to guarantee this.

7.1 Implementation notes

The basic sampling scheme followed is, first, to initialize an ensemble of models with parameters sampled evenly from across the prior, then to compress the ensemble to the posterior by applying GMC moves to the least-likely model in the ensemble at a given iterate, constrained by its current likelihood (which is the ensemble’s likelihood contour for that iterate). GMC tends to move model-particles across the parameter space very efficiently, particularly in the initial portion of compression across the uninformative bulk of the prior mass. That said, GMC particles may get “stuck” in widely separated minima, so there are instances when the least-likely particle has no valid GMC moves. In this case, the trajectory is not continued. Instead, sampling continues, unless the number of particles in the ensemble would decline below the specified minimum. In this case, a new trajectory is begun by resampling from the remaining prior mass within the ensemble. This is accomplished by random “diffusive” proposals, in contrast to the ordinary, directed Galilean movements. This means a random particle remaining within the ensemble is selected, and isotropic velocity vectors are sampled from this particle to find a starting location for the new trajectory (this vector is conferred to the new particle). If diffusional sampling fails (extraordinarily rare), a primitive ellipsoidal sampler serves as a backup. This draws an ellipsoid around the ensemble’s current models in parameter space and samples evenly from the ellipsoid. While this is an acceptable method of sampling evenly from within the existing likelihood constraint, it is wildly inefficient for posteriors with any kind of interesting topology, which is both why multi-ellipsoidal sampling exists

[FHB09], and why this single-ellipsoid sampler is confined to a backup role.

`GMC_NS.jl` follows the common convention of transforming parameter space to a unit-standardized hypersphere. The user is responsible for ensuring that the density of the prior is uniform, that is $\pi(x) = 1$, over the $-1:+1$ range of the hypersphere dimension. Utility functions `to_unit_ball` and `to_prior` which accept the prior and transformed positions and the relevant prior probability distribution, returning the unit hypersphere or prior coordinate, respectively. In future versions, this scheme is likely to be modified to a 0:1 unit hypercube, which does not require additional operations beyond obtaining the cumulative probability of a parameter value on the prior distribution. The user may also specify a sampling "box" to bound particles from illegal or nonsensical areas of the parameter space (eg. negative values for continuous distributions on positive-valued physical variables). The box reflects particles specularly in the same manner as the likelihood boundary, with the exception that because the orientation of the nth-dimensional box "side" is known (eg. is the plane at $n=0$.), the reflection is performed by stopping the particle at the box side and giving it a new velocity vector identical to the old but with reversed sign in dimension n.

`GMC_NS.jl` attempts to maintain efficient GMC sampling by per-particle PID tuning of the GMC timestep. GMC treats models as particles defined by their parameter vectors, which give their positions in parameter space, as well as a velocity vector, normally chosen isotropically and only changed when the particle "reflects" off the ensemble's likelihood contour. In GMC, when a model-particle is to be moved through the parameter space (in this case, because it is the least likely particle in the ensemble), a new position is proposed by extending some distance along its velocity vector through the parameter space. This distance is determined by scaling the velocity vector by a "timestep" value, which can be thought of as the speed with which the particle is covering parameter space. As the model ensemble is compressed down to the posterior, this timestep must decline fairly evenly in order for GMC to be efficient, as the amount of available parameter space declines rapidly. Additionally, GMC particles may enter convoluted regions of parameter space that form small likelihood-isthmuses to other, more likely regions. In order to deal with this, each trajectory is assigned its own PID tuner, which maintains an independent timestep for the trajectory, tuned to target a user-supplied unobstructed move rate. In short, this will reduce the timestep if the particle is repeatedly reflecting off the likelihood boundary (in which case it is crossing the remaining prior mass without sampling much from it, or it is in a highly convoluted area of the local likelihood surface), and extend the timestep if it repeatedly moves without encountering the boundary, so that particles that are closely sampling an open region without encountering the boundary begin to "speed up".

7.2 Ensemble, Model, and Model Record interfaces

In operation, `GMC_NS.jl` maintains an `GMC_NS_Engine` mutable struct in memory. The directory to which the sampler ought to save the ensemble is specified by its `path` field. `GMC_NS.jl` does not maintain calculated models in memory, instead serializing them to this directory. In order to track the positions and likelihoods of model-particles within the ensemble and as samples from the posterior, a `GMC_NS_Engine` maintains two vectors of `GMC_NS_Model_Records` as fields; `models` for live particles, and `posterior_samples` for the previous positions along trajectories. Observations against which models are being scored, prior distributions on model parameters, model constants (if any), and the sampling box are specified by the `GMC_NS_Engine`'s `obs`, `priors`, `constants`, and `box` fields, respectively. The

`GMC_NS_Engsemble` also specifies settings for the GMC sampler and the PID tuner. Sensible defaults for these values may be passed en masse to an ensemble constructor with the `GMC_DEFAULTS` constant exported by `GMC_NS.jl`.

Therefore, to use `GMC_NS.jl` with their model, the user must write a model-appropriate version of these three structs with the fields specified by the docstrings in the `/src/ensemble/GMC_NS_Engsemble.jl` file, as well as `/src/GMC_NS_Model.jl`. The user is responsible for an appropriate constructor and likelihood function for the model, any required parameter bounding functions, and so on. The user may optionally write a overloaded `Base.show(io::IO, m::GMC_NS_Model)` function for displaying the model, or a `Base.show(io::IO, m::GMC_NS_Model, e::GMC_NS_Engsemble)` function for displaying the model with observations. This function can be used with the package's `ProgressMeter` displays for on-line monitoring of the current most likely model.

7.3 Usage notes

7.3.1 Setting up for a run

7.3.2 Parallelization

`GMC_NS.jl` does not have an explicit parallelization scheme; the sampler proceeds linearly to the next least-likely particle and performs the appropriate Galilean trajectory sampling procedure. Parallelization may nonetheless be achieved at two levels; the likelihood function may be parallel according to the needs of the user, and individual nested sampling runs may be arbitrarily combined. That is, if one desires to parallelize a `GMC_NS.jl` job, it is best to think in terms of the total number of trajectories to be sampled, dividing this by the number of machines available to perform the sampling work, to obtain the number of particles to be sampled on each machine. The sampling runs can then be combined post hoc to achieve the desired final accuracy. In this scheme, the likelihood function is best parallelized by threading on a single machine.

7.3.3 Displays

If desired, parameters of the ensemble and most-likely model can be monitored on-line, while `GMC_NS.jl` is working. This is done by specifying a vector of function vectors (ie a `Vector{Vector{Function}}`). One such vector may be supplied to specify the displays to be shown above the `ProgressMeter`, one for those below, supplied as the `upper_displays` or `lower_displays` keyword arguments to `converge_ensemble!()`. `GMC_NS.jl` will rotate the upper and lower displays to the next vector of display functions every `disp_rot_its` sampling iterates, supplied as another `converge_ensemble!()` keyword argument. Default display functions are available in `/src/utilities/progress_displays.jl` and are exported for easy composition of the function vectors.

7.3.4 Example use

Chapter 8

BioBackgroundModels.jl

`BioBackgroundModels.jl` is a pure Julia package intended to automate the optimization and selection of large numbers ("zoos") of Hidden Markov Models, using sequence sampled from specified partitions of a given genome, on arbitrary collections of hardware. It is supplied with extensive utility functions for genomic sampling, high performance implementations of both the Baum-Welch and Churbanov-Winters algorithms for optimization of HMMs by expectation maximization (EM), as well as reporting functions for summarizing the results of the optimized model "zoos". These tasks are necessary to select of models of genomic background noise, from which motif signals may be detected using a package such as `BioMotifInference.jl`.

8.0.1 Genome partitioning and sampling

8.0.2 Optimizing BHMMs by EM algorithms

8.0.3 Displaying results

Chapter 9

BioMotifInference.jl: Independent component analysis motif inference by nested sampling for Julia

9.1 Introduction

While many methods to detect overrepresented motifs in nucleotide sequences, only one is a rigorously validated system of statistical inference that allows us to calculate the evidence for these motif models, given genomic observations: nested sampling [Ski06]. Computational biologists almost immediately benefitted from Skilling's original publication, with the release of the Java-coded nMICA by Down et al. in 2005 [DH05]. Unfortunately, this code base is no longer being maintained. It is, moreover, desireable to take advantage of modern languages and programming techniques to improve the maintainability and productivity of important bioinformatic code. We therefore re-implemented Skilling's algorithm for inference of DNA motifs in Julia [?].

9.2 Implementation of the nested sampling algorithm

Skilling's nested sampling algorithm is accompanied with an almost-certain guarantee to converge an ensemble of models on the global optimum likelihood in the parameter space[Ski06]. Skilling acknowledges at least one assumption underpinning this guarantee, which is that the sampling density (in effect, the size of the ensemble) be high enough that widely separated modes are populated by at least one model each. Although early suggestions for the generation of new models within the ensemble were generally to decorrelate existing models by Monte Carlo permutation, later work generally acknowledges that this can be highly inefficient, particularly in large parameter spaces with many modes, of which the sequence parameter spaces explored in Chapter 5 are good examples. A number of ways of addressing this problem have arisen in the cosmology and physics literature. These include methods of improving model generation, such as sampling within an ellipsoidal hypersphere encompassing the positions of the ensemble models within the parameter space [FH08, FHB09] or Galilean Monte Carlo (GMC) [Ski12], as well as methods of increasing computational efficiency, such as dynamic adjustment of ensemble size

[HHHL19].

Generally speaking, these methods are not available to us because the PWM representation of sequence signal in the ICA PWM model (IPM) is not readily susceptible to ordinary parameter space representation. This is for at least four reasons:

1. Identical IPMs may be expressed with their vector of PWM sources in different orders, so the parameter space becomes increasingly degenerate with higher numbers of sources.
2. Closely equivalent repetitive signals can be represented by PWMs on opposite ends of the parameter space (e.g. ATAT vs TATA), obviating the efficiency of ellipsoidal sampling methods and introducing a further degeneracy.
3. If model likelihoods are being calculated on the reverse strand of observations, sources on opposite ends of the parameter space (ie. reverse complements) can also represent closely equivalent signals, introducing another source of degeneracy.
4. IPM sources may be of different lengths. Even if we can relate sources in different models to one another by some consistent distance rule, it is unclear how one would decide on which dimensions of longer sources into to project the parameters of shorter ones into. BioMotifInference does maintain an index for each source against its prior, but this is no guarantee that sources remain in some way "aligned". Rather, some type of alignment would have to be done, probably massively increasing computational cost to questionable benefit¹.

These issues make it difficult to see how one would mathematically describe the position of existing ensemble models within the parameter space at all, precluding sampling from a hypersphere or the use of GMC. Therefore, BioMotifInference (BMI), like its predecessor nMICA, samples new models by applying one of a collection of permutation functions to an existing model. Additionally, although BMI is supplied prior distributions on ICA parameters from which the initial ensemble is sampled, there is no way to calculate a sensible posterior from the models generated by the nested sampling process. The primary outputs of interest are the estimation of the Bayesian evidence for model structure given the data, given as its logarithm, as well as the models in the final ensemble (which constitute samples around the posterior mode or modes).

Because the more conventional methods to address the inefficiency of nested sampling by Monte Carlo used in cosmological models described above are unavailable, BMI uses an ad hoc method of adjusting the mixture of permute patterns that are applied to models. The basic permute logic (encoded by `Permute_Instruct` arguments to the `permute_IPM` function) is as follows:

1. Select a random model from the ensemble.
2. Apply a randomly selected permutation function from the instruction's function list according to the probability weights given to the functions by the instruction's weight vector.
3. Repeat 2 until a model more likely than the ensemble contour, given observations, is found, or the instruction's `func_limit` is reached.

¹Probably some combination of dimensional analysis and fast alignment algorithms could make some headway here, but it's not clear that it's necessary to do so.

4. If the `func_limit` is reached without a new model being found, return to 1 and repeat 2 until a model more likely than the ensemble's contour is found, or until the instruction's `model_limit` is reached.

9.3 Usage notes

9.4 Recovery of spiked motifs

Part III

Supplementary Materials

Chapter 10

Supplementary materials for Chapter 2

10.1 Materials and methods

10.1.1 Zebrafish husbandry

Zebrafish used in this study were of a wild type AB genetic background. Embryos were derived from pairwise crossings. Larvae were collected upon crossing and held in a dark incubator at 28°C. At 1dpf, embryos were treated with 100 μ l of bleach diluted in 170ml of embryo medium for 3 minutes, rinsed and dechorionated. Embryo medium was changed at 3dpf. After this, all animals were maintained at 28°C on a 14-hour light/10-hour dark cycle (light intensity of 300 lux) in an automated recirculating aquaculture system (Aquaneering). Animals were reared using standard protocols[[Wes00](#)]. Fish were sacrificed by tricaine overdose at the appropriate timepoints indicated in the text. All animal experiments were performed with the approval of the University of Toronto Animal Care Committee in accordance with guidelines from the Canadian Council for Animal Care (CCAC).

10.1.2 Proliferative RPC Histochemistry

10.1.2.1 Anti-PCNA histochemistry

In order to assay the number of proliferating cells in zebrafish CMZs, we used anti-PCNA histochemical labelling, as PCNA is, in zebrafish, detectable throughout the cell cycle in proliferating cells. After sacrifice as described above, fish (or their razor-decapitated heads, in the case of fish older than 30dpf) were fixed in 1:9 37% formaldehyde:95% ethanol at room temperature (RT) for 30 minutes, followed by overnight incubation in a fridge at 4°C. Subsequently, the samples were removed from fix and washed 3 times in PBS. They were then cryoprotected by successive 30 minute rinses at RT on a rocker in 5%, 13%, 17.5%, 22%, and 30% sucrose in PBS. Samples were allowed to incubate overnight at 4°C in 30% sucrose. The next day, samples were removed from the sucrose rinse and infiltrated with 2:1 30% sucrose:OCT compound (TissueTek) for 30 minutes at RT. Samples were then embedded for cryosectioning and frozen. 14 μ m coronal cryosections were cut through the fish's heads and collected on Superfrost slides (Fisherbrand). These were stored in a freezer at -20°C until staining.

Staining was begun by allowing the slides to dry briefly at RT. Sections were outlined with a PAP pen, then rehydrated in PBS for 30 minutes at RT. Subsequently, sections were blocked in 0.2% Triton X-100 + 2% goat serum in PBS for 30 minutes at RT. This was followed by incubation in mouse monoclonal (PC10) anti-PCNA primary antibody (Sigma), diluted 1:1000 in blocking solution, at 4°C overnight. The next day, primary antibody was removed, and slides were rinsed five times in PDT (PBS + 1% DMSO + 0.1% Tween-20). Cy5-conjugated goat-anti-mouse secondary antibody (Jackson Laboratories), diluted 1:100 in blocking solution, was applied to the sections, which were incubated at 37°C for 2 hours. Secondary antibody was removed, followed by five rinses in PDT as above. Sections were counterstained with Hoechst 33258 diluted to 100 µg/mL in PBS for 15 minutes at RT. Counterstain was then removed, five rinses in PDT were performed as above, followed by a final rinse in PBS. Slides were then mounted in ProLong Gold antifade mounting medium (ThermoFisher Scientific), coverslipped, and sealed with clear nail polish. Slides were kept at 4°C until imaging.

10.1.2.2 Cumulative thymidine analogue labelling for estimation of CMZ RPC cell cycle length

In order to obtain an estimate of cell cycle length in CMZ RPCs at 3dpf, we used cumulative thymidine analogue labelling following the method of Nowakowski et al.[NLM89]. 3dpf zebrafish were divided into ten groups of n5 animals and held in 10 mM EdU for 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 8.5, 9.5, and 10.5 hours, after which they were sacrificed as described above. Samples were processed as described under “Anti-PCNA histochemistry”, except that goat-anti-mouse Cy2-conjugated secondary antibody (Jackson Laboratories) was used to label anti-PCNA, after which the Alexa Fluor 647-conjugated azide from a Click-iT kit (ThermoFisher Scientific) was added to the sections for 30 minutes at room temperature to label EdU-bearing nuclei. This was followed by five PDT rinses as described above, and resumption of the protocol with counterstaining, mounting, etc. The raw data is available in \empirical_data\cumulative_edu.xlsx

10.1.2.3 Whole retina thymidine analogue labelling of post-embryonic CMZ contributions

We used thymidine analogue labelling as an indelible marker of CMZ contributions to the retina in Fig 2.5. Because thymidine analogues are incorporated into DNA during S-phase, postmitotic progeny of proliferating RPCs which take up the label may be detected at any point after this treatment. Dosing zebrafish with a thymidine analogue for a limited period of time ensures that the label is diluted to undetectable levels in cells which continue to proliferate. Repetitive dosing thus gives rise to labelled cohorts which represent a limited set of generations of cells which become postmitotic after the dose is provided. n=3 fish were held in 10mM BrdU (Sigma) diluted in facility water for 8 hours at 30, 60, 90, 120, and 150 days post fertilisation. 10 days after the last BrdU incubation, the fish were sacrificed as described above. One representative whole retina dissection is presented.

10.1.3 Confocal microscopy and image analysis

All histochemically processed samples (coronal sections of retinas and whole retinas) were imaged on a Leica TCS SP5 II laser confocal imaging microscope, using the Leica LAS AF software for acquisition. For coronal sections, central sections were identified by their position in the ribbon of cryosections. For Fig 2.7, the central section was imaged together with the flanking section on either side of it. Confocal

data was analysed using Imaris Bitplane software (v. 7.1.0). Cell counting analyses were performed by segmenting the relevant channels using the Surfaces tool to produce counts of PCNA positive (Figs 2.7, S4 Fig) or PCNA/EdU double-positive (S4 Fig) nuclei. Lens diameters were measured in the DIC channel using the linear measurement tools, across the widest point of the section of spherical lens.

10.1.4 Estimation of CMZ annular population size

In order to estimate the total population of the CMZ in zebrafish eyes sampled throughout the first year of life, we counted PCNA-positive cells in the dorsal and ventral CMZ of central and flanking sections of these eyes as described above. We added the dorsal and ventral populations and took the average of the central and flanking section per-section populations to reduce the possibility of sampling error. We treated this as a sample of a torus mapped to the $14\mu\text{m}$ sphere segment of lens intersected by the average section. As the surface area of this zone is given by $S = \pi \times d_{lens} \times h_{section}$, and the total surface area of the lens by $A = \pi \times d_{lens}^2$, where d is the diameter of the lens and h the section thickness, we may calculate an estimate of the total population by $pop_{total} = \frac{pop_{section}}{S} * A$. We therefore obtained our annular CMZ population estimates using our measurements with the simplified formula $pop_{total} = \frac{pop_{section} \times d_{lens}}{h_{section}}$, calculating pop_{total} on a per-eye basis to account for covariance of the measurements, and subsequently averaging across n=6 eyes per sampled timepoint (3, 5, 8, 12, 17, 23, 30, 60, 90, 120, 180, and 360 dpf). These data are presented in Fig 2.7.

10.1.5 Modelling lens growth

To simulate the annular CMZ population, we wanted to be able to simulate the stem cells at the periphery of the retina occasionally undergoing symmetric divisions, so as to keep up the same density of stem cells in the first ring around the lens. We did this by normalising our lens diameter measurements to the 72hpf value, to produce a measure of relative increase in lens size over the first year of CMZ activity. We performed a linear regression of the logarithm of this relative increase vs the logarithm of time using the Python statsmodels library, using the constants derived from this regression for a power-law model of lens growth. This model was used to supply the `WanStemCellCycleModel` with a target population value as described below.

10.1.6 Estimation of 3dpf CMZ cell cycle length

To produce an estimate of the length of the cell cycle in 3dpf RPCs, we first took the total count of EdU-positive cells in dorsal and ventral CMZ from our cumulative labelling experiment described above, and divided by the total dorsal and ventral CMZ RPC population, as measured by the number of PCNA positive cells. This gave the labelled fraction measurements displayed in S4 Fig. Following Nowakowski et al. [NLM89], we performed an ordinary least squares linear regression on these data using the Python statsmodels library. We then calculated the total cell cycle time $T_c = \frac{1}{slope}$ and $T_s = T_c \times intercept_y$. While this method is not suited for heterogenous populations of proliferating cells, its use is justified here, as the He SSM to which the estimate is being applied makes similar assumptions as Nowakowski et al., in particular the homogeneity of the population. It should not be considered more than a rough estimate.

10.1.7 CHASTE Simulations

10.1.7.1 Project code

All simulations were performed in an Ubuntu 16.04 environment using the CHASTE C++ simulation package version 2017.1 (git repository at <https://chaste.cs.ox.ac.uk/git/chaste.git>). The CHASTE package is a modular simulation suite for computational biology and has been described previously[MAB⁺13]. All of the code, simulator output, and empirical data used in this paper is available in the associated CHASTE project git repository at <http://github.com/mmattocks/SMME>, as well as in the supplementary archive ???. In brief, the approach taken was to encapsulate the SSMs examined here as separate concrete child classes inheriting from the CHASTE `AbstractSimpleCellCycleModel` class. An additional model, representing the simple immortal stem cell proposed in Wan et al.[WAR⁺16], was similarly produced. Each model is generic and provided with public methods to set their parameters and output relevant simulation outcomes (and per-lineage debug data, if necessary). While these may be used in the ordinary CHASTE unit test framework, permitting their use in any kind of cell-based simulation, we wrote standalone simulator executables (apps, in CHASTE parlance) to permit Python scripting of the simulator scenarios used in this study. This allows for the multi-threaded Monte Carlo simulations performed herein. Therefore, the Python fixtures available in the project’s `python_fixtures` directory were used to operate the single-lineage simulators (`GomesSimulator`, `HeSimulator`, and `BoijeSimulator`) and single-annular-CMZ simulator (`WanSimulator`) in `apps/`, including the `CellCycleModels` and related classes in `src/`; the simulators output their data into `python_fixtures/testoutput/`. These data, along with the empirical observations (both from the Harris groups’ papers and our own described herein) available in `empirical_data/`, were processed by the analytical Python scripts in `python_fixtures/figure_plots/` to produce all of the figures used in this study.

We have attempted to make the project fully reproducible and transparent. CHASTE uses the C++ boost implementation of the Mersenne Twister random number generator (RNG) to provide cross-platform reproducibility of simulations. All of our simulators make use of this feature, permitting user-specified ranges of seeds for the RNG. We have also used the numpy library’s RandomState Mersenne Twister container to provide reproducible results for the Python fixtures, notably the SPSA optimisation fixture. The output of the project code will, therefore, be identical every time it is run, on any platform.

It should be noted that none of the code Harris’ group used in their reports has been published. We have made every effort to replicate the functional logic of the models as closely as possible. Nonetheless, it is not possible to determine precisely why, for instance, the original He SSM parameterisation failed so notably in our implementation.

10.1.7.2 SPSA optimisation of models

In order to make a fair comparison between the He SSM and our deterministic mitotic mode alternative model, we coded an implementation of the SPSA algorithm described by Spall [Spa98]. This is available in `/python_fixtures/SPSA_fixture.py`. This algorithm, properly tuned, will converge almost-surely on a Karush-Kuhn-Tucker optimum in the parameter space, given sufficient iterates, even with noisy output (eg. due to RNG noise from low numbers of Monte Carlo samples, permitting conservation of computational resources). It does so by approximating the loss function gradient around a point in parameter space, selecting two sampling locations at each iterate, then moving the next iterate’s point in parameter space “downhill” along this gradient.

Our loss function was a modified AIC; we weighted the residual sum of squares from the PD-type mitotic probabilities by 1.5 in order to improve convergence, as the PD data are the most informative part of the dataset regarding the critical phase boundaries (PP mitoses occur in all 3 phases of the He model, DD occur in phases 2 and 3, PD mitoses occur only in phase 2). We also compared model induction count output (that is, panels A-C in Fig 2.4) up to count values of 1000 to penalise parameters producing very large lineage totals.

The values of the SPSA gain sequence constants were selected to be as large as possible without resulting in instability and failure to converge; this ensured that the algorithm stepped over the widest possible range of the parameter space in finding optima. The α and γ coefficients were initially set at the noise-tolerant suggested values of .602 and .101 respectively [Spa98]. 200 iterations were performed. Initially, each iterate involved 250 seeds of each induction timepoint for the count data (panels A-C in Fig 2.4) and 100 seeds of the entire 23-72 hr simulation time for mitotic mode rate data (panels D-F). At iterate 170, these were increased to 1000 and 250 seeds, respectively, decreasing RNG noise to a low level. For the last 10 iterations, RNG noise was reduced to close to nil by increasing the seed numbers to 5000 and 1250, respectively; α and γ were set to the asymptotically optimal 1 and $\frac{1}{6}$ for these iterates, as well. The particular seeds used were kept constant throughout in order to take advantage of the improved convergence rate this affords [KSN99].

Because the simple SPSA can assign any value to parameters, our implementation uses constraints, projecting nonsensical values for parameters (e.g. negative values, mitotic mode probabilities summing to > 1) back into legal parameter space. The use of constraints has no effect on the algorithm's ability to almost-surely converge within the given parameter space [Sad97]. As we felt that the deterministic mitotic mode models' "sister shift" should be relatively small in order to be biologically plausible, we also constrained this parameter such that 95% of sister shifts would be less than the mean length of the shortest phase.

The numerical results of the SPSA algorithm are available in `/python_fixtures/testoutput/SPSA/HeSPSAOutput`, alongside png images of output from each iterate, enabling the visualisation of the progress of the algorithm.

Monte Carlo simulations Subsequent to SPSA optimisation, the parameters derived from this procedure were used to perform Monte Carlo simulations of large numbers of individual lineages, using ranges of 10000 non-overlapping seeds for each of panels A, B, C, G, H, and I, and 1000 for each of panels D, E, and F of Fig 2.4 (using the SPSA-optimised parameter sets) and [S1 Fig](#) (using the original He et al. parameterisation). This was performed using `/python_fixtures/He_output_fixture.py`.

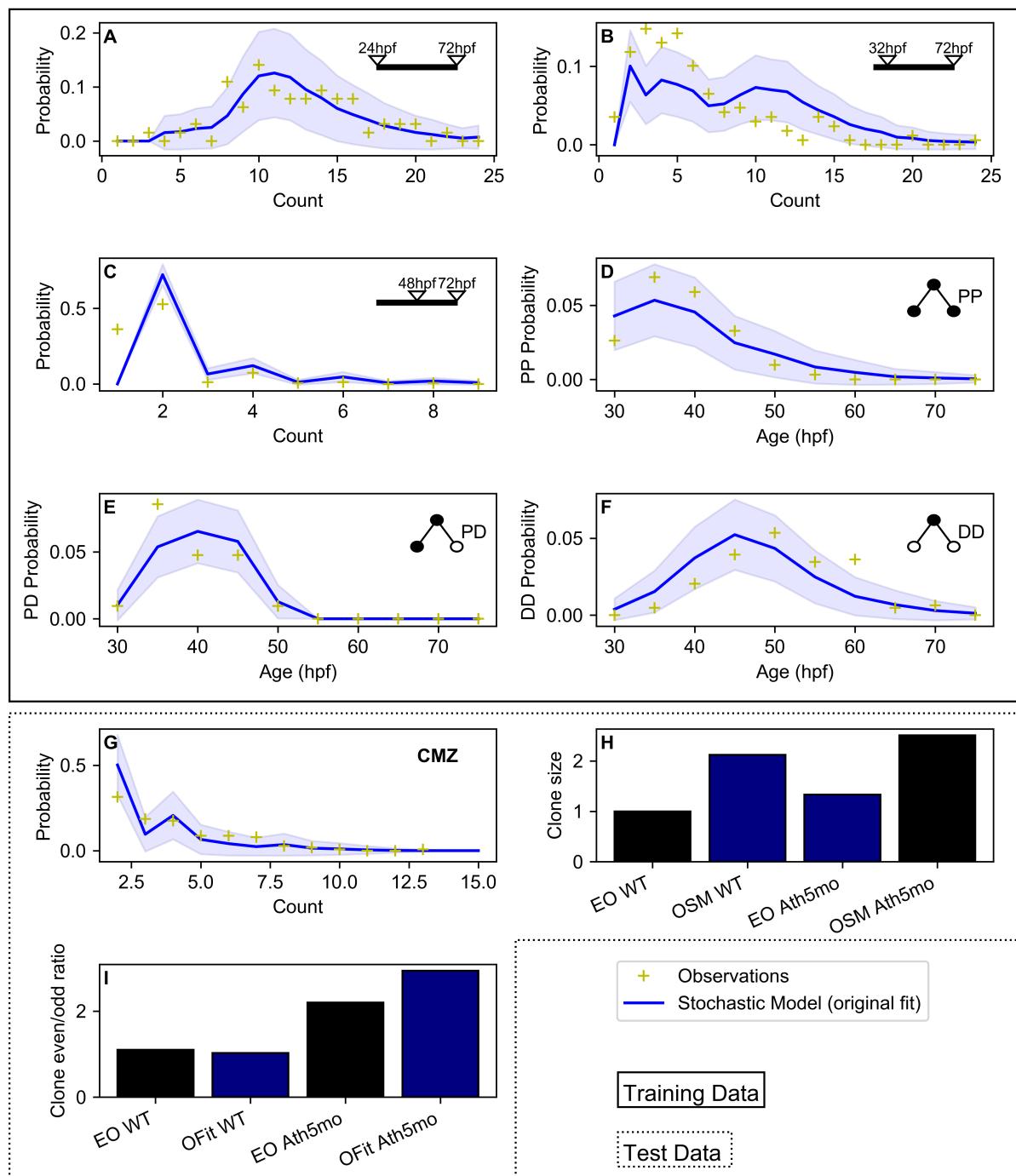
Similarly, 100 seeds were used in Monte Carlo fashion to simulate individual annular CMZ populations using `/python_fixtures/Wan_output_fixture.py`, which scripts the WanSimulator. Each of these simulations is initialised with an RPC population drawn from a normal distribution given the mean and standard deviation of the CMZ torus estimated from our 3dpf observations, as described above. Each RPC is given a TiL value randomly chosen from 0 (newly born from a stem cell) to 17hr (exiting the CMZ, "forced to differentiate", in the terms of Wan et al.). A further population of stem cells, sized at $\frac{1}{10}$ of the RPC population, is added using the `WanStemCellCycleModel`. This stem population divides asymmetrically except when it falls under its target population value determined by the model of lens growth described above; as long as this condition holds, the stem cells will divide symmetrically to keep up their numbers.

Simulation data analysis Not all of the numerical data used in He et al. and Wan et al. has been published. As such, we reconstructed the lineage data from He et al.’s Figure 5C, used by He et al. to obtain their mitotic mode probabilities; our reconstructed values are available in `/empirical_data/empirical_lineages\`. We also reconstructed the He et al. WT and Ath5 morpholino data (Fig 2.4 panels H, I) and Wan et al. lineage count probabilities (panel G) from the relevant figures. These values are entered into `/python_fixtures/figure_plots/He_output_plot.py`, where they are used in the AIC calculations described below.

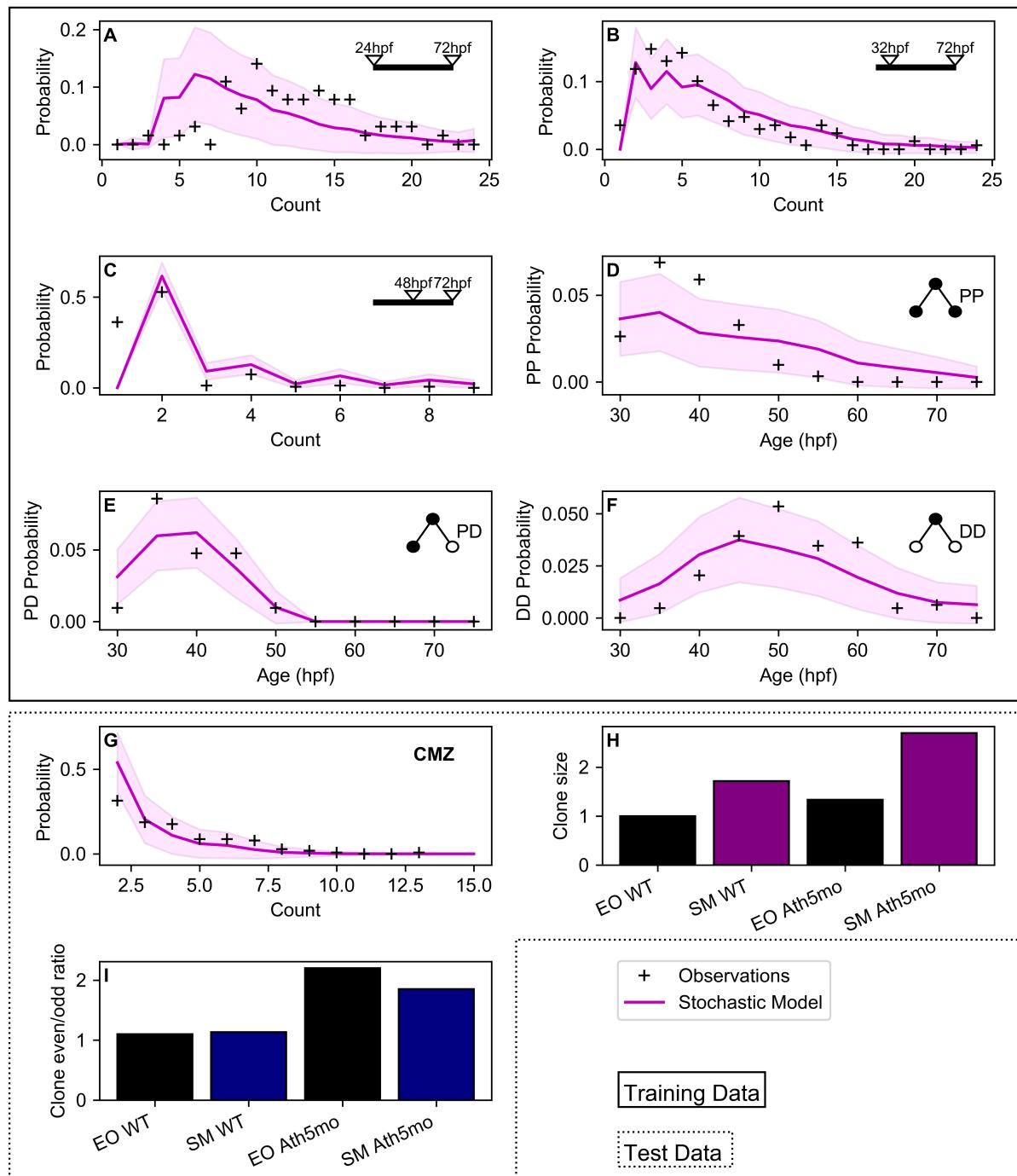
In order to assess the performance of the models used in our simulations, we used Akaike’s Information Criterion (AIC). This allows us to compare models with dissimilar numbers of parameters, as AIC trades off goodness-of-fit against parameterisation. The values reported in Table 2.1 were produced by `/python_fixtures/figure_plots/He_output_plot.py`.

The 95% CIs for the model output presented in this paper were estimated by repetitive sampling of the output data, using the same number of lineages observed empirically. 5000 such samples were performed. This provides a reasonable estimate of the confidence interval given the limited observational sample size.

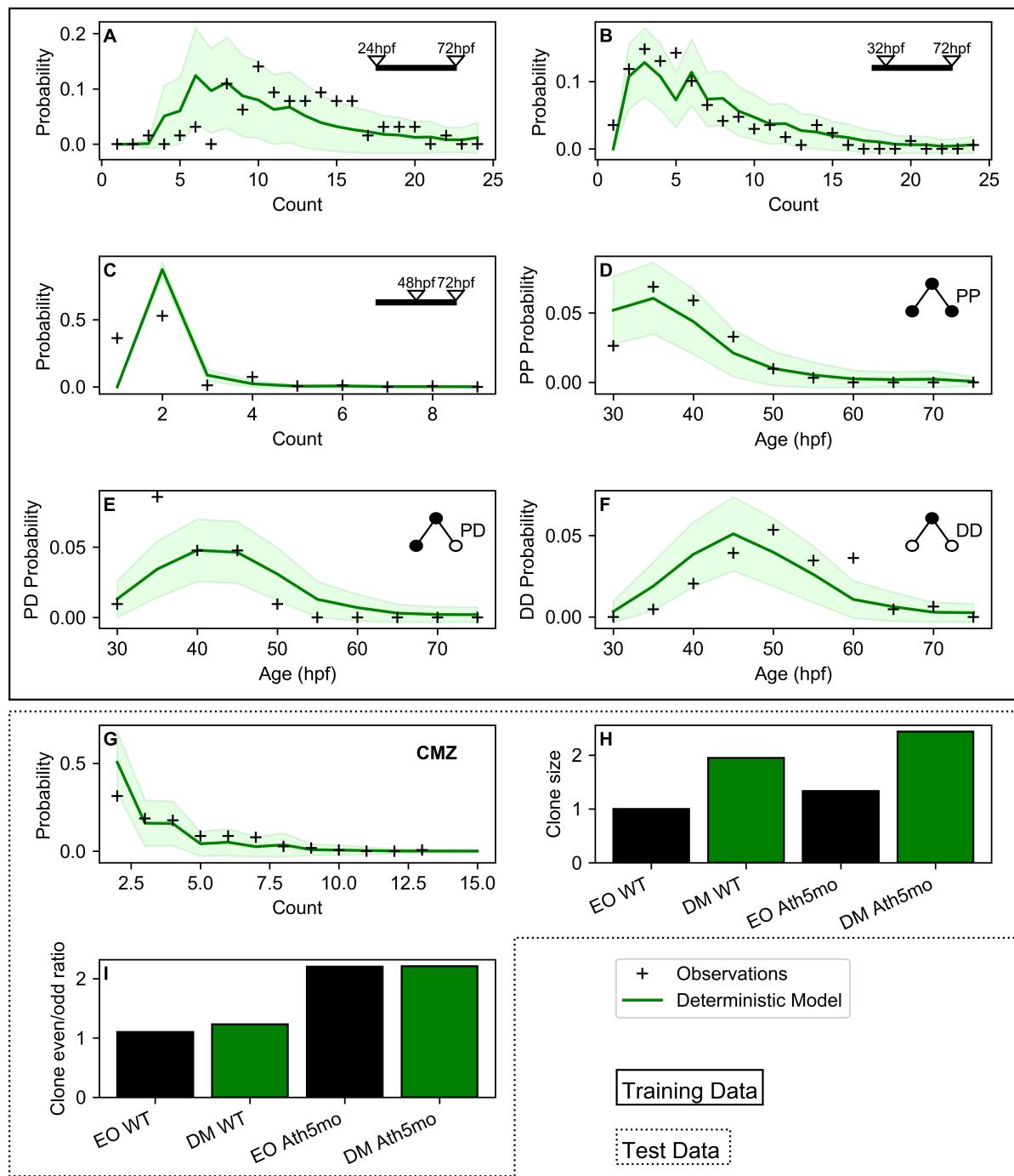
10.2 Supporting figures



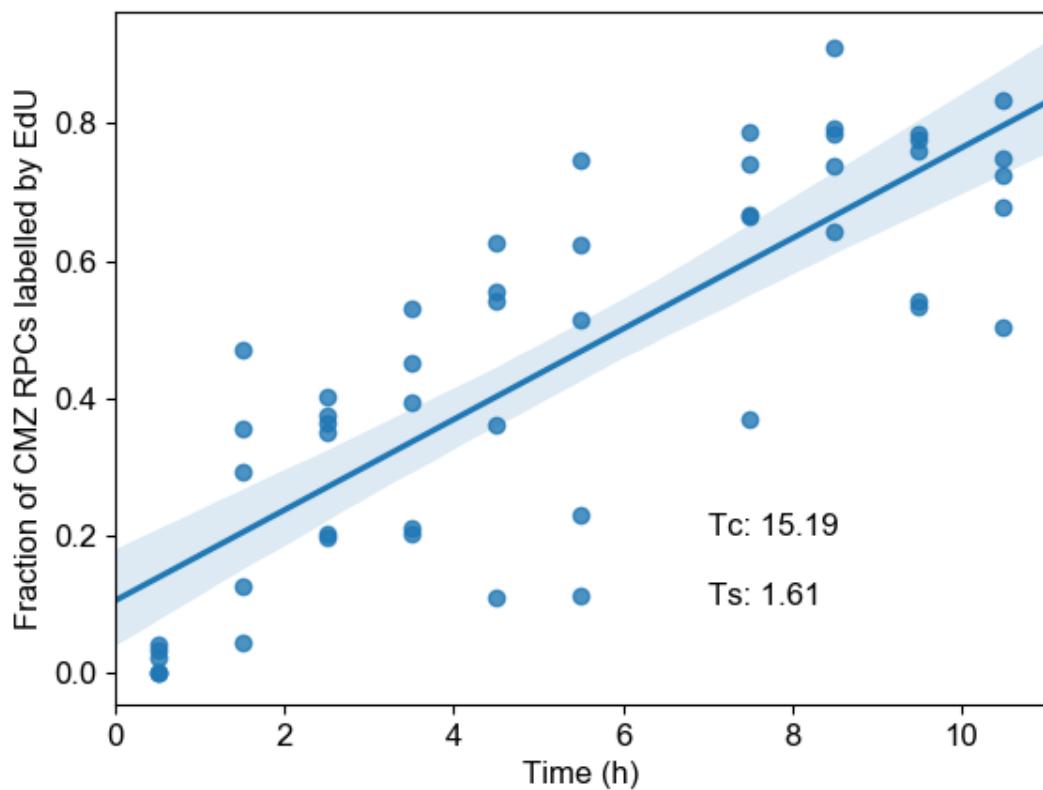
S1 Fig. He SSM original parameterisation model output. As Fig 2.4; model output uses the parameterisation given in He et al.[HZA⁺12]. Note significant deviation from observations in panel B, reflecting excess proliferative activity of modelled RPCs.



S2 Fig. SPSA-optimised He SSM model output. As Fig 2.4; stochastic model output displayed alone.



S3 Fig. SPSA-optimised deterministic mitotic mode model output. As Fig 2.4; deterministic model output displayed by itself.



S4 Fig. Cumulative EdU Labelling of 3dpf CMZ RPCs Fraction of PCNA-labelled CMZ RPCs which bear EdU label over time, as determined by histochemical labelling of central coronal cryosections of 3dpf zebrafish retinas. Dots represent results from individual retinas. Line is the ordinary least squares fit \pm 95% CI. Cell cycle length (Tc) and S-phase length (Ts) are estimated from this fit following the method of Nowakowski et al.[NLM89]

Chapter 11

Supplementary materials for Chapter 4

- 11.1 Description of the computational cluster used in the work
- 11.2 Developmental progression of naso-temporal population asymmetry in the CMZ

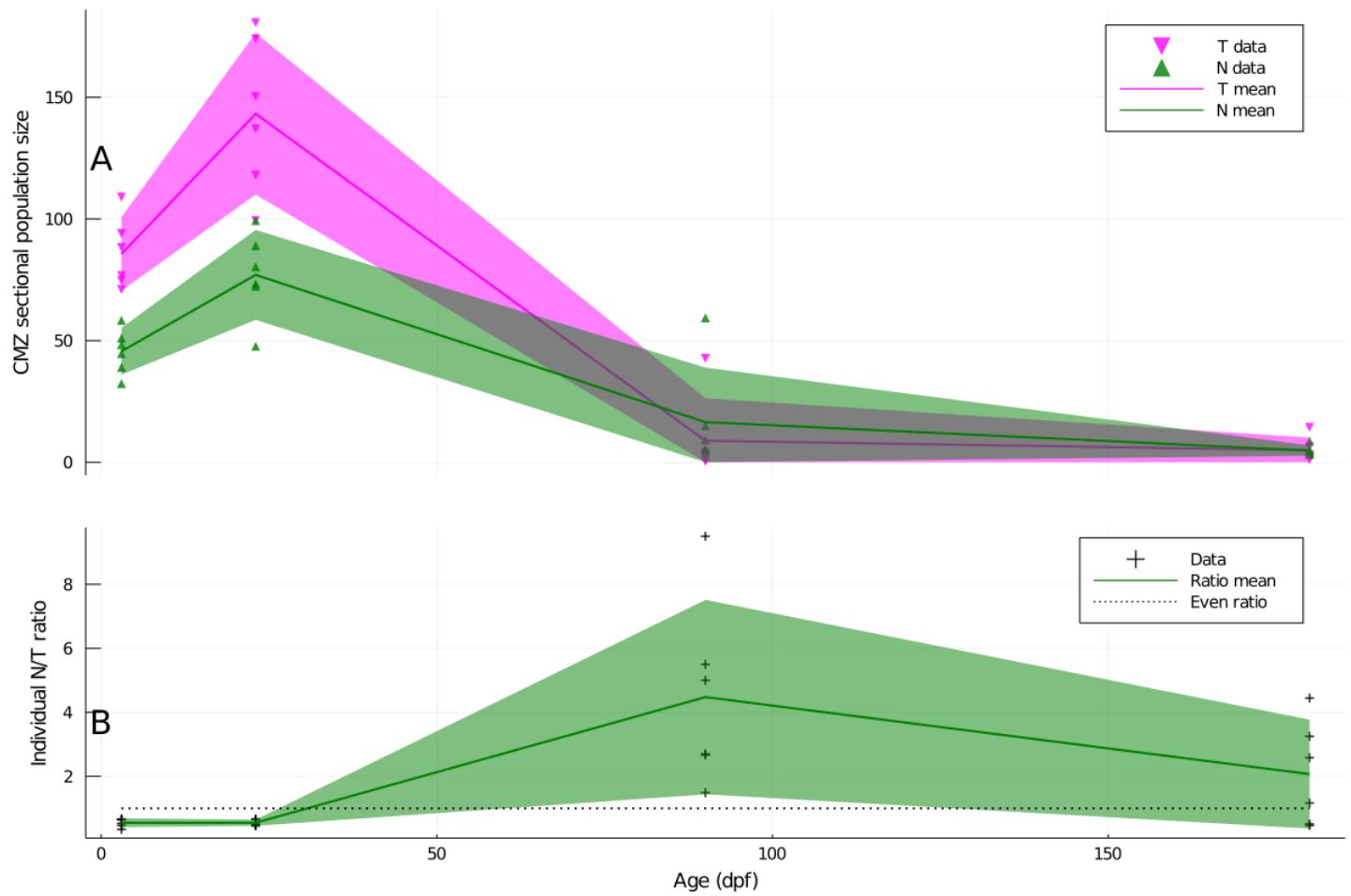


Figure 11.1: **Developmental progression of naso-temporal population asymmetry in the CMZ.**

Marginal posterior distribution of mean nasal (N) and temporal (T) population size in $14\mu\text{m}$ transverse cryosections (panel A) or intra-individual N/T count asymmetry ratio (panel B), $\pm 95\%$ credible interval, $n=6$ animals per age. Data points represent mean counts from three central sections of an experimental animal's eye.

11.3 Cumulative EdU Bayesian Linear Regression

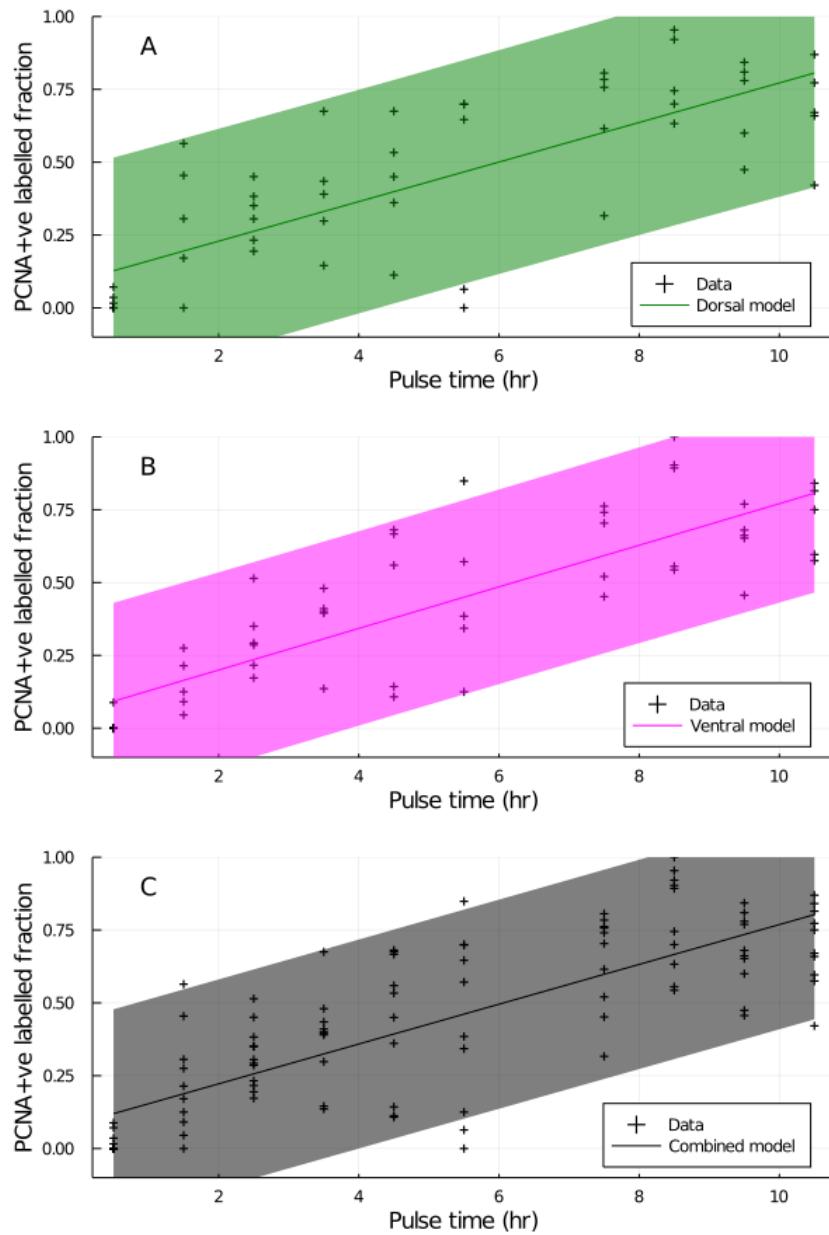


Figure 11.2: Linear regressions performed on cumulative labelling data from dorsal, ventral, and combined CMZ sectional populations

Layer	Marker	Cell type	\mathcal{N} logZ	Log- \mathcal{N} logZ	logZR	σ sign.
GCL	Cohort	All GCL cells	-189.0 ± 1.4	-193.5 ± 1.9	-4.5 ± 2.4	-1.90
GCL	Isl2b	RGC	-74.5 ± 1.0	-98.68 ± 0.24	-24.2 ± 1.1	-22.74
GCL	Pax6	Displaced am.	-101.93 ± 0.96	-77.87 ± 0.27	24.06 ± 1.0	24.15
GCL	Isl2b/Pax6	RGC subtype	-97.9 ± 1.0	-113.4 ± 0.74	-15.5 ± 1.3	-12.37
INL	Cohort	All INL cells	-184.0 ± 1.4	-474.5 ± 2.7	-290.4 ± 3.1	-94.02
INL	Pax6	Amacrine cell	-36.83 ± 0.88	-65.43 ± 0.24	-28.6 ± 0.92	-31.25
INL	PKC β	Bipolar cell	-37.83 ± 0.2	-0.77 ± 0.67	37.06 ± 0.7	53.12
INL	GS	Müller glia	-25.11 ± 0.2	6.7 ± 1.1	31.8 ± 1.1	29.22
INL	HM	Horizontal cell	-31.97 ± 0.27	6.73 ± 0.61	38.7 ± 0.66	58.37
ONL	Cohort	All ONL cells	-201.0 ± 1.5	-335.3 ± 1.5	-134.2 ± 2.1	-63.65
ONL	Zpr1	Double cones	-93.7 ± 1.4	-146.91 ± 0.8	-53.2 ± 1.6	-33.23

logZ: logarithm of p(D), the marginal likelihood of the data, or model evidence. Largest evidence values bolded. logZR: evidence ratio; positive values in favour of stable model.

Layer	Marker	Cell type	\mathcal{N} MLE	Log- \mathcal{N} MLE	lhR
GCL	Cohort	All GCL cells	94.374	91.888	-2.487
GCL	Isl2b	RGC	8.265	7.403	-0.862
GCL	Pax6	Displaced am.	8.905	11.25	2.345
GCL	Isl2b/Pax6	RGC subtype	14.464	15.248	0.784
INL	Cohort	All INL cells	75.439	71.902	-3.537
INL	Pax6	Amacrine cell	24.062	28.753	4.691
INL	PKC β	Bipolar cell	38.285	40.184	1.899
INL	GS	Müller glia	45.279	49.521	4.242
INL	HM	Horizontal cell	43.817	46.461	2.645
ONL	Cohort	All ONL cells	65.658	69.605	3.947
ONL	Zpr1	Double cones	11.934	12.782	0.848

logZ: logarithm of p(D), the marginal likelihood of the data, or model evidence. Largest evidence values bolded. logZR: evidence ratio; positive values in favour of stable model.

11.4 Evidence calculations for Normal and Log-Normal models of layer and lineage contribution

11.5 Likelihood ratio calculations for Normal and Log-Normal models of layer and lineage contribution

11.6 Methods

11.6.1 Statistical Analyses

Chapter 12

Supplementary material for Chapter 5

12.1 Synteny analysis of genomic surroundings of *D. rerio* npat

Figure 12.1 displays the output of the Synteny Database tool [CCP09] for the genomic region hosting npat on *D. rerio* chromosome 15. On the scaffold of *H. sapiens* chromosome 11, NPAT occurs upstream of the highlighted duplication cluster bracketed by ARCN1 and MIZF. Zebrafish npat is located in the midst of this rearranged, duplicated cluster, but does not appear to have been duplicated itself; at least if it was, no parologue can be found by syntenic or similarity analysis.

12.2 10dpf *rys* RPCs are mitotic

12.3 The fate of *rys* RPCs is microglial phagocytosis

12.4 PWM sources detected in the combined sib and *rys* differential position set

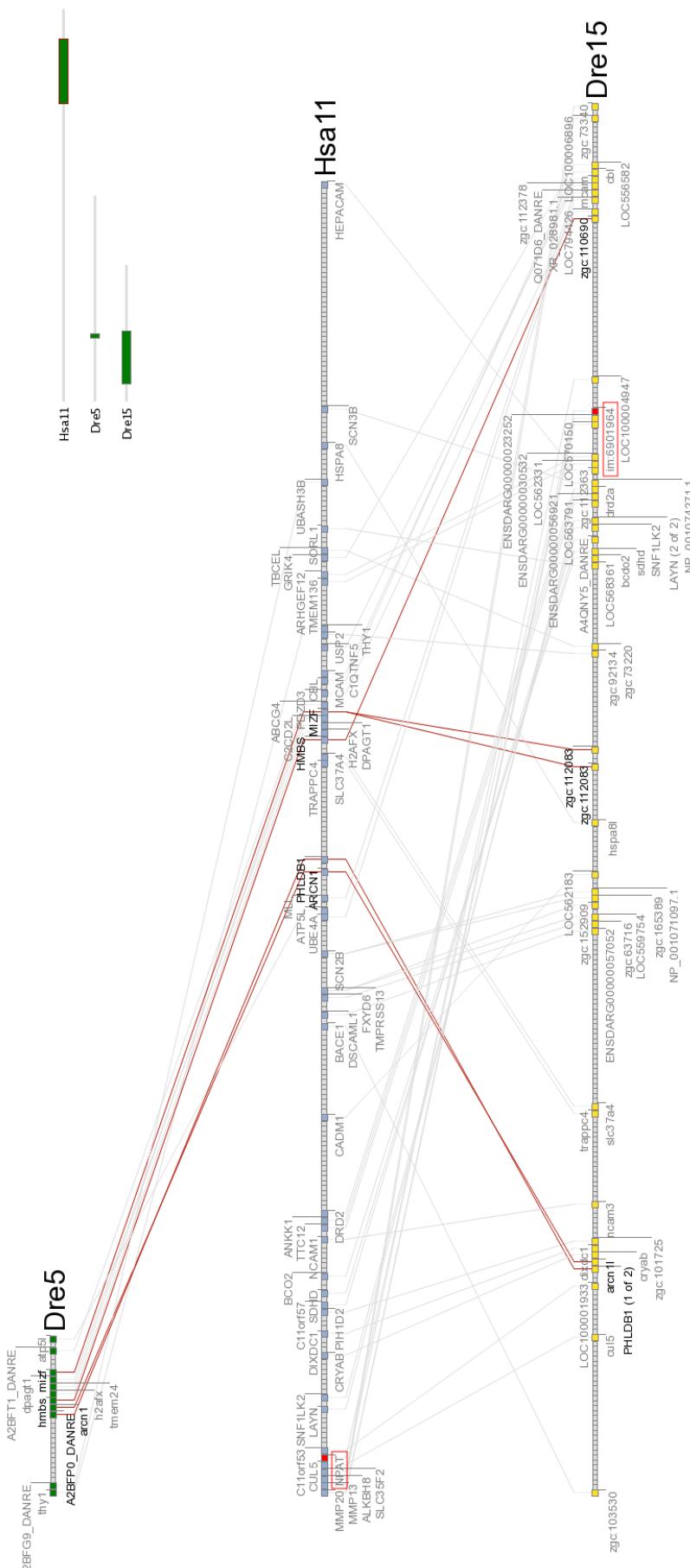


Figure 12.1: Synteny Database output for the syntenic region containing *D. rerio* npat

Dre#: *Danio rerio* chromosome# Hsa#: *Homo sapiens* chromosome #

The relative position of the displayed genomic neighbourhoods on their chromosome scaffolds is displayed inset, top right. *D. rerio* mpat is highlighted with a red square, visible on the right of the selected Dre15 region, annotated “im6901964”. *H. sapiens* NPAT is highlighted similarly on the left of Hsa11. Landmarks of the duplication and rearrangement event that brackets the position of mpat on Dre15 are highlighted with red lines.

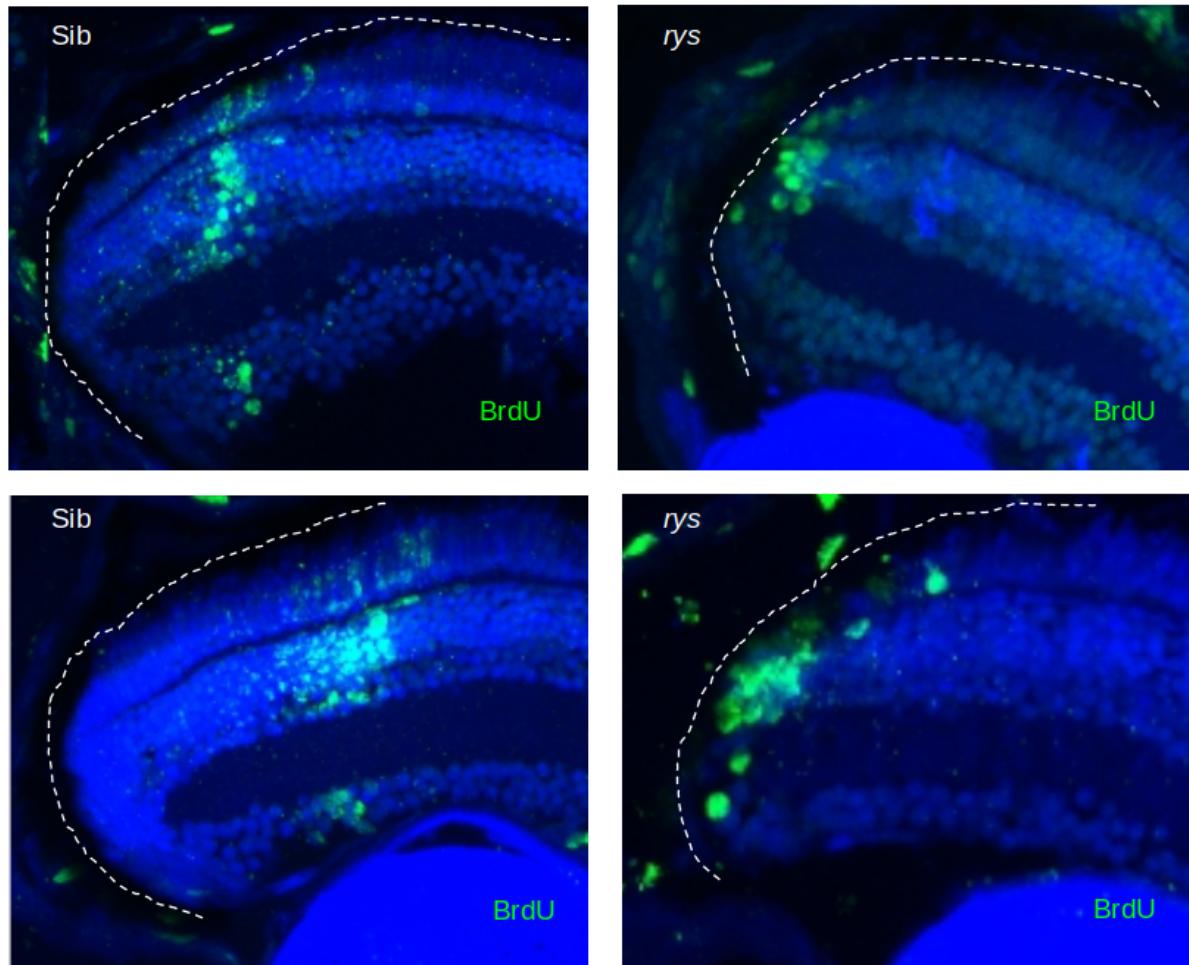


Figure 12.2: **Rys CMZ RPCs are mitotic at 10dpf**

MIP 14 μ m coronal cryosections through representative sib (left panels) and *rys* eyes at 10dpf, 7 days after an 8hr BrdU pulse at 3dpf. Note that few labelled *rys* cells have entered the specified retinal layers.

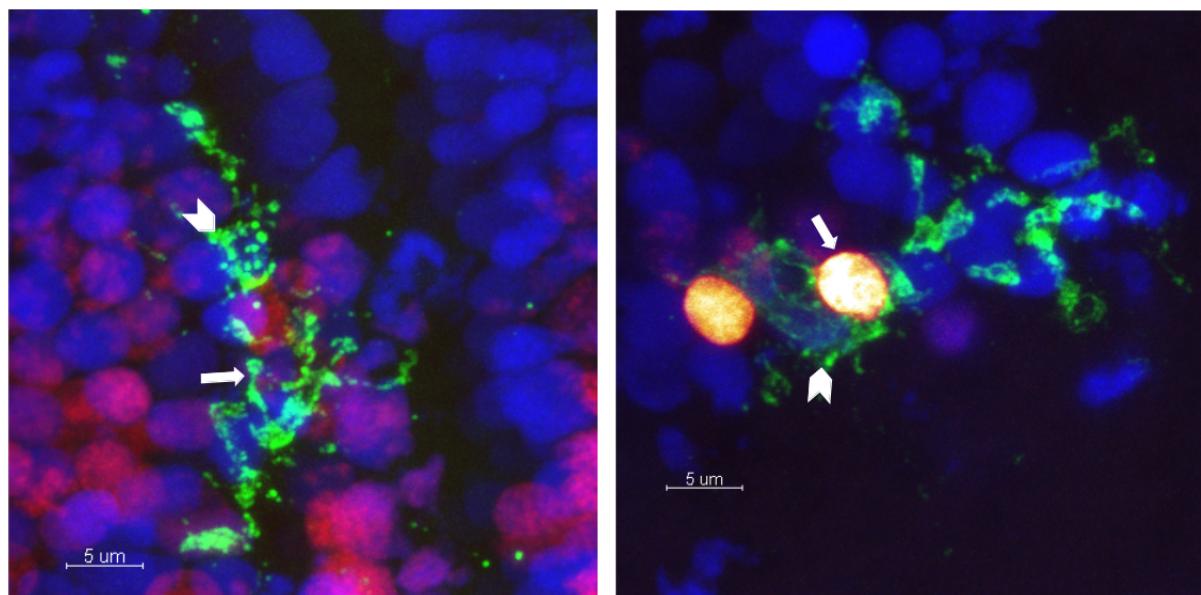


Figure 12.3: 4C4-positive microglia engulf *rys* mutant RPCs

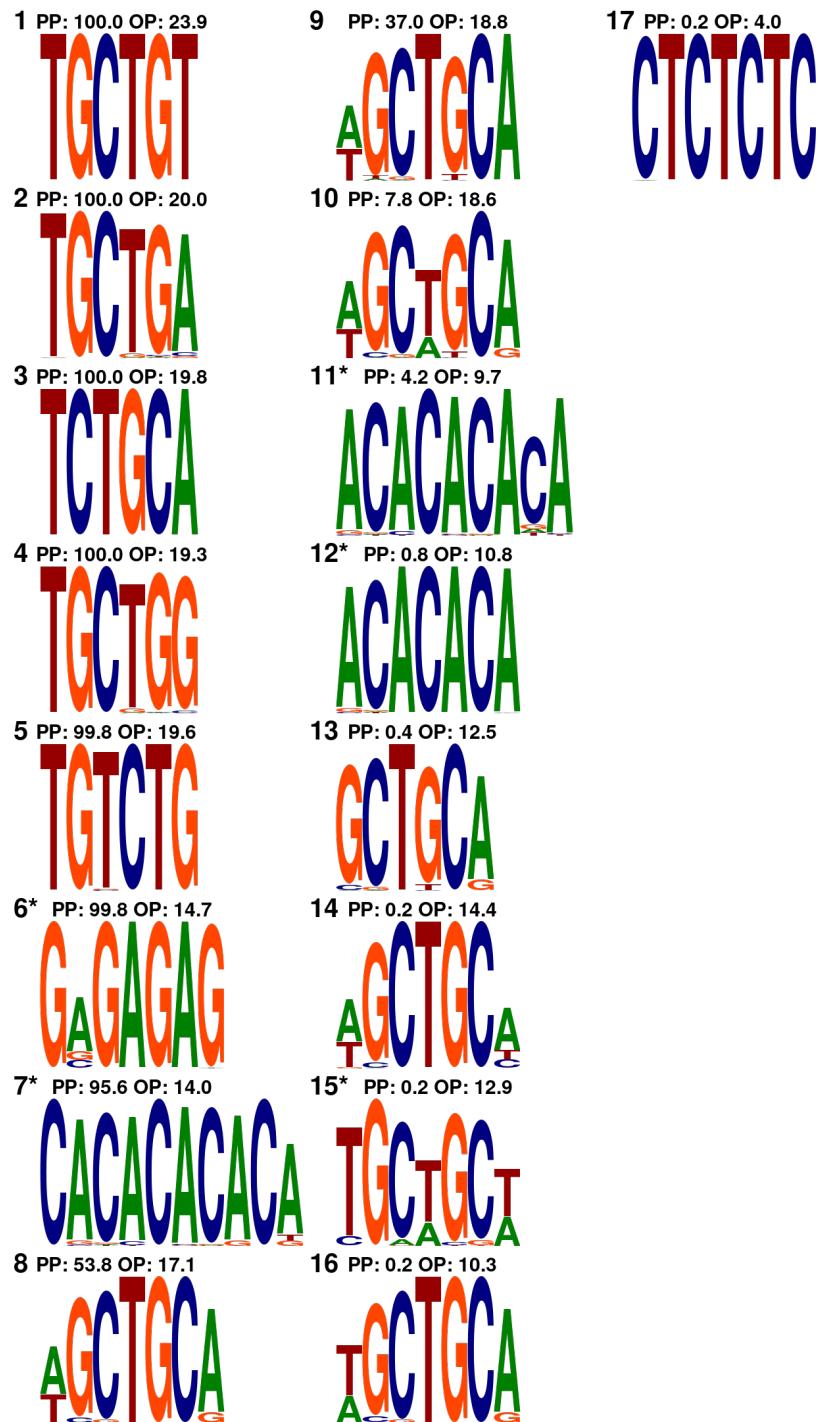


Figure 12.4: PWM sources detected in combined differential sources.

Counts of positions found in sib but not rys (red bars, represented as negative numbers, as these are ‘lost’ in rys) and those found in rys but not sib (blue bars, ‘gained’). The magnitude of the difference between the counts is represented with a yellow bar.

Chapter 13

Theoretical Appendix A: Model theory and statistical methods

13.1 Model Theory

This thesis is mainly concerned with a particular kind of scientific model: those that are mathematically expressible, and therefore tractable subjects for numerical simulation techniques. Most biological models do not fit this description, but increasing computing power has greatly expanded the potential for, in particular, cellular and macromolecular explanations to be formalized and tested in this way. While doing so is sometimes derided as “physics envy”, the reality is more complex. Sometimes, simple heuristics are adequate biological explanation; much of differential diagnosis consists of descriptive relations between qualitatively characterised signs in patients and imputed disease states. More often, the complexity of biological systems, and the sophistication of the instruments with which we probe them, give rise to observations which support more than one plausible causal explanation for the features of the data. The question of model comparison then immediately arises; without numerical analysis, we may make reference to the use of the model in practical domains where the failure of the model to reflect reality results in its disuse. Because biological models are rarely required to succeed as engineering tools for the prediction and control of outcomes of practical significance, an inability to analyse them numerically means that we are left only with rhetoric and handwaving. This type of “model comparison” necessarily devolves into the type of anarchic Feyerabendian discursive chaos discussed in Section 14.2.1. Feyerabend is correct to point out that scientific models can only be compared against one another, and that the selection of comparative criteria can never be “objective” in the sense of being independent of the contingent situation and goals of the observer. Nothing can be done about this; either we look for relatively-better criteria and relatively-better models as judged by those criteria, or, like Feyerabend, we abandon the study and practice of science for poetry.

The support of complex biological systems for often gives rise to calls for explanatory pluralism [Bri10]. It should be emphasized that very differently parameterised models can be adequate explanations for the same system. This is particularly common for explanations at different descriptive depths. For instance, the models used in Chapter 4 make no reference to particular macromolecules, but we would still like to have macromolecular explanations of RPC function, particularly because these may supply us with means to intervene onto RPCs. In other cases, adequate, differently-parameterised models of the same

system may be describing different aspects of the same level of organization. Pharmacological kinetic studies of G-protein association with receptors and crystallographic studies of the same phenomenon are a good example. These models allow experimenters to pursue different descriptive and interventional objectives. Indeed, as Nicholas Rescher has noted, attempts to synthesize many models into a single overarching explanation usually result in descriptive chaos [Res00, p.65-6]. Rescher explains how the descriptive adequacy of some model in its local domain usually requires the airbrushing out of at least some pertinent details of the system that could have been included; it can be added that the computational tractability of the model requires the same.

If we virtually all accept this form of pluralism, we are still left with the cases where models are making contradictory claims about reality¹. Pluralism cannot coherently extend to abandoning the fundamental logical law of non-contradiction, without compromising the entire endeavour of scientific rationation. We cannot accept logically contradictory notions about the structure of reality without precluding cognitive harmony with a broadly consistent view of the way the world works [Res05]. Given the complexity of biological systems, we have no option except to express models formally and to test them rigorously against one another. This is the task of model selection.

Model selection requires that we be able to score models against the phenomena they represent, as encoded by observational datasets. This requires the selection of a loss or likelihood function. Loss functions, like AIC, used in ??, express the relative amount of information in the dataset lost by the model, given a set of parameters. Likelihood functions, used elsewhere, express the likelihood of the data given the model, given the parameters. The structure of the model thus defines an n-dimensional parameter space; the loss or likelihood function expresses a surface within this space. By sampling within this space, we may estimate the shape of the surface. This sampling information can be used in three ways: we may propose new, more likely parameter space locations to sample from, in search of the least lossy/most likely parameterisation of the model (model optimisation); we may derive marginal likelihoods for particular parameter values (parameter estimation), or we may estimate the marginal probability of the model over all sampled parameterisations (evidence estimation). These topics are discussed below.

13.1.1 Bayesian Epistemological View on Model Comparison

The analyses presented in the data chapters express two views on how model comparison should be approached. The first, expressed in Chapter 2, is drawn from information theory, while the second, taken up in Chapter 4, is a relatively conventional Bayesian view, albeit with more sophisticated tools and more computing power than has been available to Bayesians in the past. In the same way that frequentist analyses may be expressed as a subset of Bayesian analyses (i.e. they normally seek to express the maximum a priori model parameterisation and likelihood from uninformative priors), informational theoretical approaches to model comparison can be expressed as a subset of Bayesian model comparison theory. In fact, the loss function used in Chapter 2, Akaike Information Criterion, has been adapted to refer to a prior distribution, as the Bayesian Information Criterion [PB04]. The intent of these criteria is to overcome the limitations of the maximum-likelihood approach by using both the maximum-likelihood value (or MAP score, in the case of BIC), and the number of parameters, to produce a score which penalizes models with more free parameters.

¹That is, they have incompatible metaphysical content; they are therefore subject to counterinduction as explained in Section 14.2.1

The general approach of optimizing a model for a loss function against a dataset, then penalizing the best model loss score by the parameterisation of the model, allows us to overcome the most important problem with frequentist approaches to model selection: the requirement for models to be parametrically nested. Model nesting fundamentally precludes counter-induction; we cannot compare the adequacy of models which express different views of how the described system is parameterised, only whether adding more parameters improves a particular view. Escaping this limitation is what allows us to compare stochastic and deterministic mitotic mode models in Chapter 2; these models express fundamentally different views of how reality is organised. Still, in important ways, this approach shares the basic problem of simply calculating the MLE: the score in no way accounts for the relative robustness of the model fits. That is, a model which is a terrible description of a dataset over most of its plausible parameter space, but an excellent one in a tiny region, will appear to be a better explanation than a model which is a broadly good description over the whole parameter space. Moreover, simply using the number of parameters to penalize the best model found in some sampling procedure fails to express the extent to which the inclusion of those parameters is justified by improving the overall likelihood of sampled models.

This leads us to what may be regarded as the completion of the Bayesian view on model selection, John Skilling's system of Bayesian inference, nested sampling [Ski06, Ski12, Ski19]. By rearranging the usual presentation of Bayes' rule, Skilling reveals how it specifies the computational inputs and outputs associated with the activities of model sampling, parameter estimation, and evidence estimation. Bayes' rule is typically written as follows, where x is a proposition about the data (e.g. specific values for the cell cycle length and exit rates of the CMZ), and $Pr(a|b)$ denotes the probability of a given b:

$$Pr(x|data) = \frac{Pr(data|x)Pr(x)}{Pr(data)}$$

This can be read aloud as "the posterior probability of the proposition, given the data, $Pr(x|data)$, is equal to the likelihood of the data, $Pr(data|x)$, given the proposition, multiplied by the prior probability of the proposition, $Pr(x)$, and divided by the marginal probability of the data over all propositions, $Pr(data)$." This gives the impression that the principal task in statistical analysis is the calculation of the posterior probability of a model, and gives rise to the treatment of the marginal probability, $Pr(data)$, as a mere normalizing constant. Skilling rearranges this to put computational inputs on the left, and outputs on the right:

$$Pr(x)Pr(data|x) = Pr(data)Pr(x|data)$$

This shows us that the evaluation of a model consists of supplying a prior probability for the proposition, $Pr(x)$, and a likelihood function to assess the probability of the data given that proposition, ($Pr(data|x)$). In return we receive, as computed output (over many samples) the total evidentiary mass of the data for this model, ($Pr(data)$), as well as the posterior parameter estimates, $Pr(x|data)$. Sample model parameters are drawn from the prior; the likelihood of this proposition about the data is calculated. By accumulating many such samples, the marginal probability of the model over all of these propositions (the evidence for the model) can be estimated. Because these samples may be weighted by their calculated likelihoods and position on the prior, we may also use them to estimate the marginal posterior probabilities of parameters of interest.

This encapsulates both the numerical procedures involved in model analysis, as well as the epistemological view implied by Bayesian statistics. That is, a model analysis is the joint product of the model and the data, which expresses our belief about the overall credibility of the model ($Pr(data)$, ie. a better model gives higher marginal probability to observations than a worse one), as well as allowing us to estimate the distribution of credibility we should assign to various values for parameters of the model (propositions), $Pr(x|data)$. These are the two fundamental levels on which quantitative measurements of natural systems allow us to make inferences. We may distinguish between models of the systems in a general sense by their evidence, when applied to the same overall dataset. This allows us to counterinductively test contradictory descriptions of the structure of the phenomenon against one another, inferring which is a better map to the territory. A second level of inference is the ranking of propositions for the parameterisation of models achieved by the sampling procedure. This allows us to determine which particular propositions about the system are supported, given the model. Because the posterior distributions need not be unimodal, we can evaluate these modes as separate hypotheses that are supported to varying degrees by the data.

This view dispenses with typical frequentist interpretations of model selection as being about finding the “true model” of reality, or of estimating the actual, objective probabilities inhering in things or processes (a view refuted in [Section 14.1](#)). Instead, we are guided to focus on the relative quality of models in explaining all of the relevant data we can gather; we may then evaluate the relative quality of specific propositions about the system within those models as we see fit.

13.1.2 Model sampling and optimization

13.1.3 Overfitting

13.1.4 Monte Carlo simulation

13.1.5 Simple Stochastic Models

The Simple Stochastic Model is schematically summarised in [Figure 13.1](#). This is the basic structure of the great majority of formal models in the stem cell literature, derived from post-hoc analyses of populations taken to include stem and progenitor cells. The population-level approach is usually explicit, as no differentiation is made between types of proliferating cell- in general, no particular cell is identified with a stem cell, nor can any be identified from the necessarily retrospective population data used to infer the parameters of the model.

The central concept of the model is that divisions can be categorised by the number of progeny which remain mitotic after the division. It is important to note that a mitotic event cannot presently be categorized in this fashion except retrospectively. This must be kept in mind when analysing models of this type, as this categorisation does not necessarily imply that there is some mechanism by which the cell specifies the fate of offspring *at the time of mitosis*, although there is extensive evidence for the coupling of mitotic and specification processes at the molecular level.

In effect, then, the model compresses the process of fate specification into individual mitotic events. Since the primary distinction between cells in the model is simply whether they are proliferating or not, the model also elides any heterogeneity within the proliferating population. Beyond not identifying particular cells as “stem cells”, this may make models derived from the SSM inappropriate for proliferative populations with a large degree of heterogeneity. One may think here of the classic idea of a

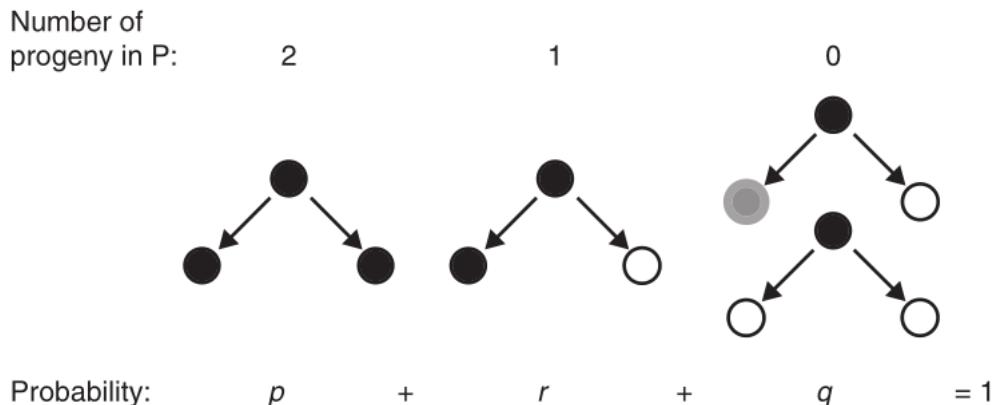


Figure 13.1: Simple stochastic stem cell model, representing probabilities of cell division events, excerpted from Fagan 2013 pg. 61. Black circles denote proliferative cells, while white and grey circles denote different types of postmitotic offspring. “Number of progeny in P” is the number of mitotic offspring produced by each type of division. The probability of each division type must sum to 1, as all possibilities are represented, granting that the division types are defined by the postdivisional mitotic history of the offspring.

small number of slowly proliferating “true” stem cells and a larger population of rapidly dividing “transit amplifying” progenitors- this type of internal structure within the proliferating population can only be represented by multiple, independent SSMs (as implemented in Chapter 4).

As Fagan notes, in the SSM, “relations among p, r, and q values entail general predictions about cell population size (growth, decrease, or ‘steady-state’), and equations that predict mean and standard deviation in population size, probability of [lineage] extinction, and features of steady-state populations are derived.”²[Fag13, p.60]

Typically, this type of model has been employed to describe population dynamics of proliferating cells in assays generating ostensibly *clonal* data, where a “clone” here refers to the population constituted by all of the offspring descended from some particular (usually “initial” and sometimes therefore taken for “stem”) proliferative cell. This population is the *lineage* generated by some particular dividing cell.

At the core of the SMME are variants of the most common model used by stem cell biologists, the *Simple Stochastic Model* or *SSM*, described in more detail in Section 13.1.5. The SSM consists of a Galton-Watson branching process, a stochastic process originally intended to model the lineage extinction of surnames. Wikipedia describes a stochastic process as “a mathematical object usually defined as a collection of random variables” [Wik18]. In the case of branching process models applied to proliferative cells, the random variable determines the mode of division of each cell within the lineage, with this mode being defined by the proliferative state (construed in the model as being either mitotic or postmitotic) of progeny. For any given division, a cell may produce two mitotic, one mitotic and one postmitotic, or two postmitotic progeny, and each of these division modes is given a defined (often, but not always, static) probability. Given these values, the history of a cell lineage may be simulated; the output of many of these simulations pooled together, usually by what is referred to as the “Monte Carlo method”, allows the statistical properties of the dynamics of population of simulated cells to be

²While Fagan refers to “stem cell” extinction, the model does not specifically define stem cells, nor does it imply intergenerational continuity, such that a particular intergenerationally identified stem cell should be said to have become extinct. The unit which survives or is made extinct is the lineage derived from some particular proliferative cell.

estimated.

The concept of “stochasticity” has a long and fraught pedigree in the SCBT. We find it deployed in an identical manner in the early 1960s as today, as in the classic modelling effort of Till, McCulloch, and Siminovitch³ [TMS64], explaining the variability in the size of ectopic spleen colonies formed by hematopoietic stem cells in irradiated mice by means of a Monte Carlo simple stochastic model (SMM). “Stochastic” is used in an ambiguous manner here, and, importantly, we find precisely the same ambiguity in Harris’ a half century later. This ambiguity arises from the application of the term “stochastic” to the biological *process* under investigation, to the process’ *outcomes*, and to the *model* constructed to describe the process. This is particularly obvious in this early work, entitled “A Stochastic Model of Stem Cell Proliferation”, which states that “variation [in clonal lineage size] may be generated by a well-known probabilistic (‘stochastic’) process, the ‘birth-and-death’ process”, and that this “process is operative when an entity, for example, a single cell, may either give rise to progeny like itself (‘birth’), or be removed in some way (‘death’) and these two events occur in a random fashion.” [TMS64] As the significance and import of the SMM directly depend on what the meaning of “stochastic” is, and since the manner in which Harris uses this concept is directly descended from the usage of Till et al., we must sort through this ambiguity before proceeding.

We may clarify the matter by returning to the biological issue at hand, which Till et al. frame in the terms of classic cybernetic control theory:

[T]he development of a colony involves processes of differentiation occurring among the progeny from a single cell. Analysis of the cell content of the resulting colony might be expected to cast light on any control mechanisms which act during colony formation. If rigid control mechanisms are operative, acting on cells of a relatively constant genotype, all colony-forming cells might be expected to behave in a similar fashion, and colony formation should be a relatively uniform process, giving rise to colonies with very similar characteristics. Alternatively, if control is lax, colonies with widely differing characteristics might be expected to develop. Results which may bear on this problem are available from experiments in which colonies were analyzed for their content of colony-forming cells. It was found that, while most colonies contained these cells, their distribution among colonies was very heterogeneous, with many colonies containing few colony-forming cells, and a few containing very many. This result suggests that control is lax. [TMS64]

This makes quite clear that the essential question here is how the process which produces stem cell proliferative and specificative behaviours is structured. We may think of Waddington’s topological model of cellular specification, itself inspired by cybernetic theory. Rigid and lax control schemes would, if modelled in this topological fashion, present themselves as very different “landscapes”. The abstract concept At the logical extreme, rigid control of stem behaviour would be represented by a single deep channel, down which the “ball” (behaving something like a ball bearing) representing “cellular state” rolls, reliably, at the same rate, in every single instance. Conversely, extremely lax control would be represented by a landscape resembling a Galton board⁴, a broad, flat slope with many pegs which may bump the ball this way or that, tending to cluster the balls in the general center of the slope but never preventing some of them from ending up at either extreme (a properly constructed Galton board

³It is worth noting that none of these investigators actually performed the relevant modelling calculations, leaving these to the U of T computer scientist L. Cseh. This division of labour remains lamentably common, and is likely responsible for many of the problems biologists have in understanding the meaning of their mathematical models.

⁴The Galton board, named after its inventor, Sir Francis Galton, is significant because the statistical methods Till et al. use to solve the ‘birth-and-death’ process were also invented by him.

produces a Gaussian distribution of balls at the bottom of the slope). This metaphor immediately reveals that, *contra* Till et al.'s interpretation of

13.2 Statistical Methods

13.2.1 Bayesian parameter estimation

13.2.1.1 Problems with frequentist inference using normal models of sample data

Typical biological practice is to report the mean and variance of a sample, assuming a normal distribution of the error around the mean. In other words, the sample is taken to be representative of a larger population; that population is modelled by a normal distribution with mean μ and variance σ ; the parameters of the maximum likelihood estimate (MLE) for the normal model are the values reported. A slightly more sophisticated approach is to report the standard error of the mean of the sampling distribution the sample is taken to be drawn from, if more than one sample can be obtained, although this usually plays no role in hypothesis testing (often conducted by t-test).

This approach has a number of defects which follow from one another. We are reporting MLE parameters without any account of our uncertainty about those parameters. There is no way to incorporate prior information we have about the parameters (even just to admit total ignorance about them). This leads to *overfitting* of our estimates to the sample data. Practically speaking, this means our estimate of the mean is stated too precisely, and the variance is too sensitive to outliers.

Additionally, the plain-sense interpretation of the estimates are often unclear. Means are usually reported plus-minus variance, $\mu \pm \sigma$, and σ is often erroneously interpreted as uncertainty about μ rather than an estimate of a second parameter, the variance of the normal population model. If the frequentist confidence interval for μ is reported, it is explicitly not understood as the interval in which we have e.g. 95% confidence that μ lies, but rather as the interval in which, in the case we repeat the experiment indefinitely, μ will be found in 95% of samples. Hypothesis tests are given similarly confusing interpretations involving long-run repeated experiments. These interpretations are widely, if not ubiquitously, misunderstood or ignored in favour of technically incorrect but comprehensible ones [?, ?].

13.2.1.2 The Bayesian approach to normal models of unknown mean and variance

Bayesian methods rectify these problems by understanding the normal model as a model of our information about the population and not of the population itself. This epistemological view of statistics is explicated in ?. Normal gaussian distributions are well-justified both by their ubiquitous success in parameter estimation and by information theoretic considerations [JBE03], and need not reflect the actual distribution of the population. However, we wish to express our uncertainty about the parameters of a normal gaussian distribution by giving further distributions over the mean m and variance of the normal distribution, with variance usually expressed as precision, $\lambda = 1/\sigma$. Typically, this is done by assuming normally distributed uncertainty on μ and gamma distributed uncertainty on λ , giving rise to a joint normal-gamma (NG) distribution [?]:

An NG distribution may thus serve as a model of our prior information about the population being measured. Because my estimates are the first ones I have made about the relevant populations, and I

have no specific guide as to the actual numbers of cells to expect, I have chosen to use the uninformative NG prior:

$$p(m, \lambda)$$

lies within that range, but is rather understood as the probability that, if the experiment were repeated indefinitely, 95

The appropriateness of the normal model is often in question because it is taken to represent some actually-existing population (which are often not well modelled by normal gaussians). Comparisons of these models using t-tests are given complex interpretations involving long-run rates of error

In Bayesian statistics, available information about a parameter is often modelled by a gaussian distribution over possible values of the parameter.

Chapter 14

Theoretical Appendix B: Metaphysical Arguments

14.1 Chance is not a valid explanation for biological phenomena; Randomness is a measure of property of sequences and not an explanation

Substantial portions of this document are dedicated to an examination of Harris' regular invocation of "stochastic" and related adjectives to describe the behaviour of RPCs. This is what lead me to characterise the theory primarily in those terms- the Stochastic Mitotic Mode Explanation. It may not be immediately obvious why this should be the case; most scientists assume that they know what words like "stochastic" and "random" mean well enough to use them in rigorous technical publications. We may not be aware that there has been a sprawling debate on the meaning of these terms since the earliest statistical formulations began to appear in the 19th century. However, even the simplest examples (as current today as they were in Laplace's time) reveal how difficult this topic can be.

If we consider the classic example of the coin flip, a process whose outcome we generally regard as being in some way "stochastic" or due to "chance", we immediately face the question of whether these descriptions refer to our inability to know the outcome of the process, or whether they refer to properties of the process itself. In other words, if we could specify the mechanics of the coin toss with sufficient precision, could we predict the outcome? This reflects two possible senses in which we may legitimately describe a process as "stochastic": referring to an epistemic dimension (we may describe some process as stochastic because we are unable to predict its outcome *a priori*), or referring to an ontological dimension (we describe the process as stochastic because this, in some way, describes how it *really is* independent of our knowledge of it).

Complicating matters is the sheer number of implications that we tend to associate with "stochasticity" and "randomness". We may be saying something about the causal structure of an event with deep metaphysical implications. It is common to distinguish between "deterministic" and "stochastic" processes, as though "stochastic" literally meant "indeterministic"- something like the Copenhagen interpretation of quantum physics. We may mean something about the apparent disorderliness of a series of outcomes of some process, with mathematical and information theoretical implications. What is an

apparently simple observation- cellular fate distribution in RPC lineages is “stochastic”, now seems to require at least a little clarification or interpretation. In general, we may say that stochasticity, for Harris, applies to at least the following entities:

1. The population-level phenomenal outcome of the RPC fate specification process (the phenomenon itself)
2. The overall behaviour of the macromolecular system whose operations produces these outcomes (the macromolecular mechanism itself)
3. The particular behaviour of some component of the macromolecular system, eg. stochastic expression of transcription factors (the macromolecules themselves)
4. The statistical generalisations used to characterise relevant aspects of the behaviour of the mechanism or the macromolecules

It is, moreover, hardly fair to expect Harris to be advancing a coherent theory about the ontological, objective basis of randomness or probability. Still, this leaves us in the awkward position of not knowing quite what the leading explanation for retinal formation is actually saying about its explanandum. The SMME is thus at risk of circularity- the explanandum (unpredictable variability in clonal outcomes) has as explanans a mechanistic explanation containing an abstract mathematical model tuned to produce this unpredictable variability. This may, in other words, turn out to be a convoluted case of model overfitting, if the “stochasticity” in question does not have a material biological referent. Before considering this, we need to define our terms more carefully to avoid the pervasive confusion mentioned above.

14.1.0.1 Chance versus Randomness

A commonplace belief is that randomness refers to outcomes produced by chance events. In an extensive and useful discussion, Antony Eagle reviews the evidence for this Commonplace Thesis, or (CT)[Eag18], drawing on discussions in the physics literature. Importantly, he notes that chance and randomness are not identical, and that one can conceivably exist without the other. This, in effect, disproves the (CT)- it is very difficult to imagine how the two concepts can be directly related in this productive fashion. I will attempt a brief summary of Eagle’s argument, with reference to Kolmogorov complexity:

Chance is mainly used to refer to processes. Exemplars are coin flips and die rolls. We can think of these as “single-case” probabilities that we take to inhere in the process. For instance, we may say that an evenly weighted coin has a .5 probability of returning a value of heads on a flip, even if it is only flipped once. That is, probabilities can be taken to be objective properties of individual instances of processes, and not only descriptions of the frequencies of the process’ outcomes over many repetitions. This is closely related to the logical concept of “possibility”. If something is possible, it has a chance of occurring. However, possibility is a logical binary; something is either possible or impossible. A “single-case” probability is understood as something like an objective feature of a system as a whole given its actual configuration and the relevant natural laws.

Randomness, by contrast, mainly refers to process *outcomes*. That is, randomness is a property of a series of outcomes of multiple instances of some process. It turns out to be challenging merely to define what a “random” binary sequence might be (perhaps generated by a series of coin flips). However, in general, we may say that a random sequence of outcomes is one that cannot be generated

by an description shorter than the sequence itself. That is, there is no set of rules that can generate a genuinely random sequence from a shorter sequence. In algorithmic information theory, the length of the ruleset required to produce some piece of information (like a sequence of measured outcomes) is called the Kolmogorov complexity of that object; if the Kolmogorov complexity of the object is equal to the object's length, the object definitionally has the property of algorithmic or Kolmogorov randomness¹.

Eagle produces numerous examples of the dissociability of these concepts, from which I have selected two concise illustrations:

Chance Without Randomness

...

A fair coin, tossed 1000 times, has a positive chance of landing heads more than 700 times. But any outcome sequence of 1000 tosses which contains more than 700 heads will be compressible (long runs of heads are common enough to be exploited by an efficient coding algorithm, and 1000 outcomes is long enough to swamp the constants involved in defining the universal prefix-free Kolmogorov complexity). So any such outcome sequence will not be random, even though it quite easily could come about by chance.

...

Randomness Without Chance

...

Open or dissipative systems, those which are not confined to a state space region of constant energy, are one much studied class [of the objects of deterministic classical physics- notably, biological systems are dissipative], because such systems are paradigms of chaotic systems ... the behaviour of a chaotic system will be intuitively random ... [t]he sensitive dependence on initial conditions means that, no matter how accurate our finite discrimination of the initial state of a given chaotic system is, there will exist states indiscriminable from the initial state (and so consistent with our knowledge of the initial state), but which would diverge arbitrarily far from the actual evolution of the system. No matter, then, how well we know the initial condition (as long as we do not have infinite powers of discrimination)², there is another state the system could be in for all we know that will evolve to a discriminably different future condition. Since this divergence happens relatively quickly, the system is unable to be predicted ... Just as before, the classical physical theory underlying the dynamics of these chaotic systems is one in which probability does not feature. [Eag18]

Therefore, the (CT) is untenable. Processes are “chancy”; collections of process outcomes, “trials”, or instantiations are “random”. It is tempting to say that Harris is explaining random fate outcomes with descriptions of chancy processes occurring internally to RPCs. Let us examine whether this is plausible.

14.1.0.2 Chance in molecular mechanisms

I turn first to consider what it might mean to describe the behaviour of a biological macromolecular system as “chancy”. Let us again distinguish between the ontological and epistemic dimensions of this description. There is a sense in which the operation of a macromolecular mechanism could be said to

¹The Kolmogorov complexity of a sequence can be estimated, contrary to common belief[LV08]. Minimum Message Length expresses a similar concept. More prosaically, one may simply compress a serialized representation of the sequence on one's hard drive with a reasonably good compression algorithm; the degree of compression achieved is a good indicator of the degree of non-random order available to shorten the sequence's description.

²Note that this condition defines chaotic randomness as an epistemic, rather than ontological, feature of complex systems- a being with infinite powers of discrimination could predict the evolution of a complex classical system with perfect accuracy.

be objectively chancy, and one in which the “chancy” outcome reflects our ignorance of some source of variability in the process.

Eagle proffers two common lines of argument in favour of chanciness as an objective property of processes. The first is the notion of the “single-case” probability mentioned above. The examples given are single coin flips, and the decay of single radioactive atoms, which are commonly taken to have chancy outcomes irrespective of anyone’s beliefs about them. As Eagle notes, this is closely related to frequentist ideas about stable processes, or trials:

It is the stable trial principle that has the closest connection with single-case chance, however. For in requiring that duplicate trials should receive the same chances, it is natural to take the chance to be grounded in the properties of that trial, plus the laws of nature. It is quite conceivable that the same laws could obtain even if that kind of trial has only one instance, and the very same chances should be assigned in that situation. But then there are well-defined chances even though that type of event occurs only once.

...

The upshot of this discussion is that chance is a *process* notion, rather than being entirely determined by features of the outcome to which the surface grammar of chance ascriptions assigns the chance. For if there can be a single-case chance of $\frac{1}{2}$ for a coin to land heads on a toss even if there is only one actual toss, and it lands tails, then surely the chance cannot be fixed by properties of the outcome ‘lands heads’, as that outcome does not exist. The chance must rather be grounded in features of the process that can produce the outcome: the coin-tossing trial, including the mass distribution of the coin and the details of how it is tossed, in this case, plus the background conditions and laws that govern the trial. Whether or not an event happens by chance is a feature of the process that produced it, not the event itself. The fact that a coin lands heads does not fix that the coin landed heads by chance, because if it was simply placed heads up, as opposed to tossed in a normal fashion, we have the same outcome not by chance. Sometimes features of the outcome event cannot be readily separated from the features of its causes that characterise the process by means of which it was produced.

[Eag18]

Examining the example of the coin toss, we find a fairly simple answer to the question posed earlier: if we knew enough about the mechanics of the toss, could we predict its outcome? The answer is yes, we can- the Bell Labs statistician Persi Diaconis has built a coin tossing machine that reliably produces heads or tails [Kes04]. We therefore know that tightly controlling the mechanics of a coin toss allows us to treat this system as entirely deterministic, without any significant element of chance in the outcome. A coin toss is only chancy when the human doing it does not have full control over the mechanical parameters of the process. Conceptually, there is no *a priori* reason why a coin-tosser should not be able to regularise the angular momentum of their thumb-flick by training with a strain gauge, place the coin on a stable surface allowing flicking, and achieve the same effect as the coin-tossing machine. In this case, Eagle’s suggestion that “[s]ometimes features of the outcome event cannot be readily separated from the features of its causes that characterise the process” seems obviously wrong- the “chancy” element of coin tossing is fully separable from the rest of the coin tossing process, and replaceable with a non-chancy component.

If the foregoing argument is correct, it seems that the coin toss is an example of the epistemic, rather than ontological, dimension of chance. The process appears to be chancy, or random, because the human tossing the coin is not able to precisely control the mechanical parameters of the process. Indeed, as

Diaconis notes, these epistemically-limited tossers do not actually produce unbiased random outcomes—human coin tosses come up as they were started slightly more often than with the obverse face [DHM07].

Eagle’s second example of an “objective single-case chance”, is the decay of a radioactive atom. This is a common method of making covert appeals to the second line of argument for objective chance, which is the existence of orthodox quantum theory. There is no known physical process whose parameters are thought to define the lifetime of individual radioactive atoms, in the way that there is a well-specified physical process that produces a particular coin toss outcome. Rather, this is an appeal to the Copenhagen theoretical principle that it is *a priori* impossible to predict the lifetimes of individual atoms. As appeals to quantum theory to ground “objective chance” in biological processes are becoming more common, let us consider whether a quantum theoretical explanation might plausibly underpin the “stochasticity” of RPC fate specification.

14.1.0.3 Can macromolecular chanciness be rooted in quantum indeterminacy?

Eagle suggests that, because the Copenhagen interpretation of quantum physics has wide currency among physicists, the theory’s implied indeterminacy of physical phenomena at the quantum level could ground “objective chance”. While common, this argument downplays the fact that quantum theory is not a homogenous scientific tradition. Unfortunately, a significant misrepresentation of the history of quantum theory has given rise to the impression that the Copenhagen theory is the unanimous or best articulation of quantum theory. We must briefly examine this misrepresentation before we can understand whether Eagle’s argument makes sense.

The conventional history of mid 20th-century quantum theory holds that John Stewart Bell, in the demonstration of his famous inequality, conclusively proved that deterministic (so-called “hidden variable”) theories of quantum mechanics were incorrect. As demonstrated (strangely, without any acknowledgement) in another section of the very pages Eagle’s argument appears in, this is a highly partisan and misleading view. Eminent Bohmian theorist Sheldon Goldstein conclusively demonstrates that Bell was an advocate of the deterministic Bohmian mechanical theory, and thought his famous inequality demonstrated that quantum phenomena could not be *local*, not that they could not be *deterministic*[Gol17].

Indeed, Bohmian quantum mechanics are fully deterministic, describe all of the same phenomena as Copenhagen, and in several cases resolve problems that orthodox quantum theory cannot [Gol17]. We are not, therefore, facing unanimous expert consensus that there is objective chance at the quantum level. We rather have a situation where physical phenomena are described by two different theory-sets, one of which takes its statistical generalisations to be descriptions of ontological indeterminacy (Copenhagen), and the other to reflect epistemic uncertainty about a determinate universe (Bohm). Moreover, there is no reason to prefer the Copenhagen approach, given that the Bohmian theory explains Copenhagen’s paradoxical results “without further ado”[Gol17]³, deals with empirically verified phenomena of physical and biological interest that Copenhagen does not (eg. electron tunneling), and was the preferred approach of the man who understood better than anyone his own results, J.S. Bell.

Therefore, Eagle’s argument is incorrect. There is no reason to suppose that the existence of quantum theoretical models that posit objective chance is good evidence for the *reality* of objective chance.

³Bizarrely, Copenhagen partisans claim that Bohmian mechanics is formally equivalent to the Copenhagen approach. If this is the case, chance is *clearly* a function of model choices and not of any underlying ontological reality. However, Bohmian mechanics is, in fact, substantially more complete than its Copenhagen equivalent, which, by Eagle’s (defective) logic, suggests reality is more likely to reflect the deterministic rather than the chancy approach.

Moreover, there are good reasons to suppose that the converse is true. In sum, then, we may say that there is no reason to consider Harris' argument to refer to *objective, ontological* chance, since the arguments for the existence of both single-case objective chance or quantum chance are weak and biologically irrelevant. Clearly, however, the *epistemological* dimension of chance is in play here.

14.1.0.4 Randomness in RPC fate specification

Having dealt with how the concept of chance might apply to Harris' SMME above, let us consider how the term "random" might relate to the process of RPC fate specification and differentiation. As introduced earlier, the technical meaning of "randomness" pertains to sequences of process outcomes. The process outcomes Harris is concerned with are the temporally-arranged fate outcomes of some particular RPC lineage. Therefore, we must ask whether these sequences meet any reasonable technical definition of "random".

Harris' own model proves that RPC fate outcomes are not algorithmically random. That is, the sequence of outcomes has a structure that can be meaningfully compressed by rules which produce typical RPC fate outcomes (Harris' mathematical models are such rule sets). One might object that Harris' meaning is that the particular rules which give rise to cellular fate "choice" in his models involve random number generation. In this case, the claim is trivially about the model and not about the sequence of outcomes that is actually observed in zebrafish eyes. Indeed, all of Harris' later models *axiomatically assume* a tripartite temporal structure to the differentiation process⁴. This is precisely the type of sequential bias which allows efficient compression of a non-random sequence of outcomes by an algorithm. Therefore, Harris himself concedes that RPC fate specification is not an algorithmically random process⁵.

We should further note that the question of how predictably ordered, i.e. non-random processes like fate specification arise in biological systems is a fundamental question of the biological sciences. It has long been recognised that classical and quantum physical systems which have algorithmically random initial conditions do not spontaneously evolve to a state of order. In many ways, then, it is the extent to which variable sequences of outcomes like RPC fate specification *depart* from algorithmic randomness which of interest when we are asking questions like "how does the ordered structure of a retina arise from RPC activity?"

14.1.0.5 Summary: "Stochastic" or "variable"?

Above, I argue that the RPC fate specification process is not objectively chancy (since objective chance is an empirically unsupported concept), nor random (since the sequence of RPC lineage outcomes is structured and therefore non-random). What, then, should we make of the argument that this process is "stochastic"? Let us consider the forceful argument of the great Bayesian statistician Edwin Thompson Jaynes:

"Belief in the existence of 'stochastic processes' in the real world; i.e. that the property of being 'stochastic' rather than 'deterministic' is a real physical property of a process, that exists independently of human information, is [an] example of the mind projection fallacy:

⁴That is, an early bias in RPC production is produced in these models by the a priori commitment to a "rule" which results in early RPC production.

⁵Having debunked the lay sense of "random" being equivalent to "chancy" above, there is no reason to consider these other, confused, non-technical definitions of randomness. They are neither logically defensible nor practically quantifiable, and therefore are of no scientific interest.

attributing one's own ignorance to Nature instead. The current literature of probability theory is full of claims to the effect that a 'Gaussian random process' is fully determined by its first and second moments. If it were made clear that this is only the defining property for an abstract mathematical model, there could be no objection to this; but it is always presented in verbiage that implies that one is describing an objectively true property of a real physical process. To one who believes such a thing literally, there could be no motivation to investigate the causes more deeply than noting the first and second moments, and so the real processes at work might never be discovered. This is not only irrational because one is throwing away the very information that is essential to understand the physical process; if carried into practice it can have disastrous consequences. Indeed, there is no such thing as a 'stochastic process' in the sense that the individual events have no specific causes." [JBE03]

It is important to emphasize that the utility of stochastic modelling techniques should not be taken to suggest that on some level the modelled phenomenon is actually, i.e. irreducibly, random and without causal structure. When speaking of biological "randomness" or "stochasticity", biologists rarely precisely define what is meant by these terms. This vagueness sometimes arises from, or results in, a theoretical deficit where properties of statistical models are understood to directly reflect the system being modelled; the scientist has failed to heed Korbzysky's dictum insisting that "a map *is not* the territory it represents, but, if correct, it has a *similar structure* to the territory, which accounts for its usefulness" [Kor05] (italics in original). The "structural similarity" here is between the model's outcomes and the collection of actually-observed population outcomes, *not* the underlying biological process giving rise to measured outcomes. I conclude by suggesting that this particular example applies to all such explanations in the biological sciences. There is presently no good reason to accept scientific explanations rooted in chance. As for explanations rooted in randomness, to be credible, these must actually estimate the degree of randomness present in the relevant outcomes, and explain departures from pure incompressibility.

14.2 Macromolecular mechanistic explanations in the Systems era

The data chapters of this thesis are concerned with two issues in scientific practice: the comparison of models with different structures, and their interpretation. I began my career in stem cell biology with a background in molecular pharmacology, and the particular explanatory worldview that training inscribed. For me, biological explanation was about elucidating mechanisms, which consisted of descriptions of macromolecules and their accessory small molecule messengers interacting. "Models," in the sense of numerical simulations susceptible to formal statistical testing, were only the formal encoding of a body of knowledge that was first proved out at the bench, in interventional experiments. That these results could be encoded by a system of differential equations (SODE) only confirmed the rigour of the original analysis which composed the mechanistic explanation in the first place. That engineers should bring their (allegedly) sophisticated numerical techniques into our laboratories, only to have their expert analyses confirm the plain-sense interpretations of the benchworkers producing the data, served to reinforce the sense that the pharmacological approach was fundamentally the correct one. It hardly ever entered our minds that model comparison was not occurring at all; it seemed that null hypotheses had already been slain by the flurry of t-tests and ANOVAe applied to the underlying datasets before the eggheads showed up.

That there were problems with this approach was already apparent by the beginning of my graduate

education; Faculty of Medicine graduate pharmacology seminars in the early 2000s routinely included dark warnings about the collapse of antidepressant effect sizes over time, and the need for new statistical approaches. Reports that most biomedical research results are not replicable were met with a sort of palpable relief [Ioa05], if not much practical change. Ultimately, the power of the macromolecular mechanism in pharmacology proved overstated; the productivity lull of the 2000-2010 era has been overcome not by rational mechanistic design of traditional small molecules, but by an influx of new classes of drugs, often with unknown or poorly characterised mechanisms of action [Mun19]. The arrival of new statistics has not helped matters much; serious Bayesian analyses tend to doubt whether medical pharmacological interventions are effective at all [Ste18]. Still, by the time I had left for greener pastures with the stem cell biologists at the new Department of Cell and Systems Biology, it was clear that the pharmacological view of biological explanation had won significant discursive battles.

A fascinating artefact of this discursive victory is present in the best available study on the use of mechanistic explanations in stem cell biology, Melinda Fagan's "Philosophy of stem cell biology: knowledge in flesh and blood". In concluding chapters outlining the future of stem cell biology, Fagan provides a diagrammatic summary of what she takes to be the field's consensus on its general direction into its future "Systems Biology" incarnation, which I reproduce here in ??.

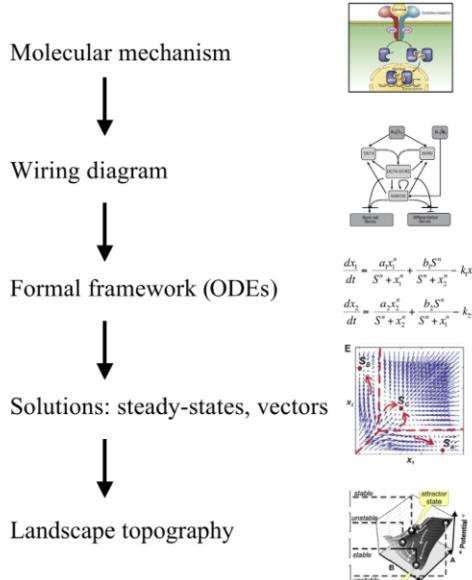


Figure 14.1: Cellular systems model construction, excerpted from [Fag15, p.7]. The system of equations and subsequent steps are based on a 2-element wiring diagram.

The perspicacious reader will observe that this is nothing other than the pharmacological view I outline above: the practice of molecular biology is the elucidation of mechanisms, the practice of systems biology is the formalization of these mechanisms as systems of differential equations. What was most peculiar about the presence of this view is that it makes no account of the Simple Stochastic Model, which receives significant coverage elsewhere in the book. This was particularly important to me because I was trying to understand how to use Harris' models to explain my results. What Harris was doing was clearly intended to be both a mechanistic explanation, in the sense that it eventually nominated a pair of particular transcription factors as the causative agents, and also a Systems explanation, in the sense that it relied, for persuasive effect, on complex numerical modelling techniques. It was formulated in SSM terms, however, not as a SODE, and critically was about the interpretation of a mathematical construct, not about the joint action of the named macromolecules, as Fagan suggests we should understand mechanistic explanations [Fag13].

Moreover, much of the sophisticated literature on mathematical models of stem cell variability are based on interpretations of dynamical systems theory or chaos theory [FK12, HLZQ17] that deal with the macromolecular constituents of cells as bulk expression values; that is, a protein

may be represented in one of these models as a coordinate on a dimension representing the amount of the protein expressed. It is extraordinarily rare for these models to be very concerned about crystal

structures, phosphorylation states, or any of the other mechanistic details the pharmacological view is so concerned with. Other approaches rely on control systems theory [SK15, YSK15], many others gaily use technical information-theoretic terms like “noise” without any effort to define or measure it [CHB⁺08], still others treat the expression of macromolecules as the endpoints of tissue-mechanical forces [PLM⁺17], and so on. The actual variety of “systems” explanatory strategies found “in the field” is bewildering; to become proficient in even one is a years-long project. As I began to grasp the sheer number of technical and theoretical subdomains implicated in these explanations, the full force of Nicholas Reschers’ argument on this point became apparent:

The ramifications and implications of philosophical contentions do not respect a discipline’s taxonomic boundaries. And we all too easily risk losing sight of this interconnectedness when we pursue the technicalities of a narrow subdomain. In actuality, the stance we take on questions in one domain will generally have substantial implications and ramifications for very different issues in other, seeming unrelated domains. And this is exactly why systematization is so important in philosophy - because the way we *do* answer some questions will have limiting repercussions for the way we *can* answer others. We cannot emplace our philosophical convictions into conveniently delineated compartments in the comfortable expectation that what we maintain in one area of the field will have no unwelcome implications for what we are inclined to maintain in others. [Res05, p.97]

Indeed, the sudden injection of the philosophical contents of the “complexity sciences” into biological discourse felt like an invasion; suddenly, it was not very clear what a mechanism or a biological explanation might consist of after all. Michel Morange has argued persuasively that molecular biology is essentially mature [Mor03] and that “systems” explanations consist mainly in the complementation of ordinary molecular practice with sophisticated mathematical models [Mor08], while acknowledging the increasing space available for the entry of physical explanations in molecular biology [Mor11]. This account is appealing, but it belies the chaotic nature of the recent scientific scene, and cannot make sense of why there are so many contending, often quite incompatible, views on how to explain cellular and molecular biological phenomena and why so few of those views include any idea on *how those explanations might be tested against one another*.

14.2.1 The Feyerabendian modeller

Chapter 2 cites Paul Feyerabend in establishing that scientific theories proceed by making metaphysical claims about reality. This point informs the model-analysis in that chapter, highlighting the sense in which the entities to which the Stochastic Mitotic Mode Explanation (SMME) refer become progressively more abstract and restricted to the mitotic mode concept. Mitotic mode is not a physical existent, but rather an abstraction compounding the fate of multiple cells- it is in this sense plainly meta-physical. Feyerabend has a number of essential insights on scientific metaphysics, which, when taken on board, allow us to make sense of what is happening within stem cell biology and more broadly, in the “systems biology” era. The central, seminal insight of Paul Feyerabend’s *Against Method* is that the observed historical succession of scientific theories occurs by counterpositional advancement of incompatible opposing theories, because only counterinductive comparisons *between* theories are capable of showing up their implicit assumptions and allowing them to be challenged. Feyerabend explains:

... it emerges that the evidence that might refute a theory can often be unearthed only with the help of an incompatible alternative: the advice (which goes back to Newton and which is still very popular today) to use alternatives only when refutations have already discredited

the orthodox theory puts the cart before the horse. Also, some of the most important formal properties of a theory are found by contrast, and not by analysis. A scientist who wishes to maximize the empirical content of the views he holds and who wants to understand them as clearly as he possibly can must therefore introduce other views; that is, he must adopt a *pluralistic methodology*. He must compare ideas with other ideas rather than with 'experience' and he must try to improve rather than discard the views that have failed in the competition.[Fey93, p.20]

Feyerabend is interested in debunking the notion that science differs from other discursive social practices by showing that no universal method vouchsafes the progressive replacement of earlier, inferior theories with later, improved ones. He famously establishes that Galileo used a form of metaphysical propaganda, particularly in the ad hoc invocation of the now-discredited concept of circular inertia, to establish the credibility of the Copernican model against its orthodox Aristotlean competitor championed by the Catholic Church. Although we commonly think of the so-called Copernican Revolution as the replacement of an obviously defective theological explanation by a properly formed theory from the empirical sciences, Feyerabend shows that this conceals the actual means by which Galileo makes his persuasive case for the (itself badly defective) Copernican model. Galileo's theory substantially contradicted the available evidence, erroneously asserted the reliability of some telescopic observations and discredited others⁶, made extensive use of ad hoc hypotheses, and was advanced by propagandistic and even dishonest means. Much of this was unavoidable. Ad hoc hypotheses are necessary for new theories because the auxiliary sciences associated with them have not been developed- scientific development is intrinsically uneven, obligating the use of these makeshift theoretical devices. Without a certain level of dishonesty and rhetorical sleight of hand on Galileo's part, the Copernican program would have succumbed to the greater development and argumentative weight of the scholastic tradition. As Feyerabend notes, if any of the typically suggested criteria for a universal scientific method were applied, the Church would have won the debate and we might still have an Aristotlean cosmology. Much the same can be said in the case of Einstein or Bohr's scientific programmes, both scientists viewing themselves as outsiders attacking an established orthodoxy.

Because of his concern to attack what he sees as the overweening social influence of scientific and technical experts, Feyerabend puts a great deal of emphasis on the discursive process of establishing orthodoxy within a field, and the sense in which this is common to scientific and nonscientific domains. That is, the practice of the natural sciences are susceptible to the same irreducibly subjective and historically contingent social, political, economic, etc. forces as all other domains of human culture, and in this sense there is nothing special about scientific practice that shields its conclusions from error introduced by these means. These extra-scientific forces have significant effects on the forms scientific models take and the interpretations they are given, and it helps to remember this when reading modelling papers published to, in effect, keep the lights on. Still, I suggest that it is important not to interpret these arguments too pessimistically. Feyerabend genuinely wanted to promote what he called "Open Exchange" between different scientific and metaphysical traditions, centered around a noncoercive exchange of, in effect, models and standards for judging them. He was also wrong that there is nothing particular to the practice of natural sciences that is not present in other domains of human activity; statistical descriptions of collections of outcomes and of uncertainty, and a self-reinforcing dynamic of scaffolding methodological complexity, are unique to the modern natural sciences.

⁶Galileo asserted that comets are optical illusions, for instance, since their non-circular orbits disconfirm the Copernican system, which insists on the circularity of orbital motion.

Ultimately, the value in Feyerabend's perspective is seeing that scientists routinely operate as epistemological anarchists, finding what works in their local institutional surroundings, and that we must expect that this will be the case. In other words, Fagan's (extremely carefully argued) conceptualization of the orthodox molecular mechanism, as found in stem cell biology, is useless for interpreting Harris' theories. This is so because Harris does not care about adhering to this standard of orthodoxy. The field has, in typical anarchic fashion, begun adapting models from a panoply of engineering, physics, statistical mechanics, AI, etc. subdisciplines, more or less willy nilly, and often without any sort of analysis of the adequacy of the model relative to alternatives. Moreover, these models pertain to a variety of levels of biological organisation, from the tissue down to subcellular nuclear compartments, and frequently do not refer in any concrete way to biological macromolecules.

I argue that, from this point of view, the macromolecular mechanistic explanation Fagan reifies is, in fact, already dead. The era of painstaking, effector-by-effector construction of macromolecular pathways, to be virtually enshrined in their ultimate SODE incarnation, perhaps ultimately to be assembled into some Monodian model cell-as-cybernetic-factory, is over, if it ever existed. This is not to say that Morange is wrong- as he asserts, molecular biology is not dead, it has not been replaced by "systems biology". In practice, however, the traditional form of molecular biological explanation is rapidly being replaced by explanatory forms from other fields, and it is not always clear that this has been salutary. For Feyerabend, science, like other human social practices consists of a variety of interacting traditions with differing assumptions, methods, sensory interpretations, and so on. The development of science is thus "not the interaction of a practice with something different and external, *but the development of one tradition under the impact of others.*" [Fey93, p.232]. The concept of "systems biology" is fundamentally an artefact of the impact of advanced statistics and statistical mechanics on biology. The resolution to the question of how this impact is mediated will determine the institutional and intellectual landscape that results from it.

To have any agency in this process, molecular biologists must be able to assess the theoretical contents that are being imported into our field for ourselves. If we do not, we cannot assess the impact of the metaphysical commitments of these theories on the rest of our theorising, as Rescher persuasively insists that we must. Indeed, it was Feyerabend's perspective that led me to realise that there were at least three scientific orthodoxies hindering me from understanding the metaphysical contents of Harris' theories. These were the mechanism-to-SODE pipeline instilled by my pharmacological training and recapitulated by Fagan, the mendacious insistence on the actual reality of chanciness by Copenhagen theorists, and the broken and illogical statistical approaches commanded by frequentist statisticians. By realising that any conceptual element could appear in a mechanistic explanation, by rejecting the objective reality of chance, and embracing the original formulation of statistical orthodoxy suggested by LaPlace (that is, Bayesian statistics), I was able to counter-inductively show up the fundamental problems with Harris' theory.

Moreover, by the conscious adoption of Bayesian methods from cosmology to address local problems in stem cell biology, I attempt to model Feyerabend's Open Exchange between two scientific traditions. My selection of Skilling's nested sampling system of inference is driven by my appreciation for its logical consistency, simplicity, deep roots in fundamental logic and information theory, and broad applicability to a wide range of problems. While Feyerabend is no doubt correct that, ultimately, the selection of criteria to distinguish between scientific theories must itself be subjective, the deep comportment of Skilling's system with the fundamental human drive toward cognitive harmony [Res05] compels me to

suggest that it may serve as a near-universal yardstick of the adequacy of scientific models with respect to their underlying datasets in the not-too-distant future.

Chapter 15

Code Appendix

15.1 SPMM

15.1.1 /apps/src/BoijeSimulator.cpp

```
 1 #include <iostream>
 2 #include <string>
 3
 4 #include <cxxtest/TestSuite.h>
 5 #include "ExecutableSupport.hpp"
 6 #include "Exception.hpp"
 7 #include "PetscTools.hpp"
 8 #include "PetscException.hpp"
 9
10 #include "BoijeCellCycleModel.hpp"
11 #include "OffLatticeSimulationPropertyStop.hpp"
12
13 #include "AbstractCellBasedTestSuite.hpp"
14
15 #include "WildTypeCellMutationState.hpp"
16 #include "TransitCellProliferativeType.hpp"
17 #include "DifferentiatedCellProliferativeType.hpp"
18 #include "BoijeRetinalNeuralFates.hpp"
19
20 #include "CellsGenerator.hpp"
21 #include "HoneycombMeshGenerator.hpp"
22 #include "NodesOnlyMesh.hpp"
23 #include "NodeBasedCellPopulation.hpp"
24 #include "VertexBasedCellPopulation.hpp"
25
26 #include "ColumnDataWriter.hpp"
```

```
27
28 int main(int argc, char *argv[])
29 {
30     ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
31     //main() returns code indicating sim run success or failure mode
32     int exit_code = ExecutableSupport::EXIT_OK;
33
34     if (argc != 13)
35     {
36         ExecutableSupport::PrintError(
37             "Wrong arguments for simulator.\nUsage (replace<=> with
38             → values, pass bools as 0 or 1):\n BoijeSimulator
39             → <directoryString> <filenameString>
40             → <outputModeUnsigned(0=counts,1=events,2=sequence)>
41             → <debugOutputBool> <startSeedUnsigned>
42             → <endSeedUnsigned> <endGenerationUnsigned>
43             → <phase2GenerationUnsigned> <phase3GenerationUnsigned>
44             → <pAtoh7Double(0-1)> <pPtf1aDouble(0-1)>
45             → <pngDouble(0-1)>",
46             true);
47         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
48     }
49     return exit_code;
50 }
51
52 /*****
53 * SIMULATOR PARAMETERS
54 *****/
55 std::string directoryString, filenameString;
56 int outputMode; //0 = counts; 1 = mitotic mode events; 2 = mitotic
57     → mode sequence sampling
58 bool debugOutput;
59 unsigned startSeed, endSeed, endGeneration, phase2Generation,
     → phase3Generation;
60 double pAtoh7, pPtf1a, png; //stochastic model parameters
61
62 //PARSE ARGUMENTS
63 directoryString = argv[1];
64 filenameString = argv[2];
65 outputMode = std::stoi(argv[3]);
66 debugOutput = std::stoul(argv[4]);
67 startSeed = std::stoul(argv[5]);
68 endSeed = std::stoul(argv[6]);
69 endGeneration = std::stoul(argv[7]);
```

```
60     phase2Generation = std::stoul(argv[8]);
61     phase3Generation = std::stoul(argv[9]);
62     pAtoh7 = std::stod(argv[10]);
63     pPtf1a = std::stod(argv[11]);
64     png = std::stod(argv[12]);
65
66     /*****
67      * PARAMETER/ARGUMENT SANITY CHECK
68      *****/
69     bool sane = 1;
70
71     if (outputMode != 0 && outputMode != 1 && outputMode != 2)
72     {
73         ExecutableSupport::PrintError(
74             "Bad outputMode (argument 3). Must be 0 (counts) 1
75             ↳ (mitotic events) or 2 (sequence sampling)");
76         sane = 0;
77     }
78
79     if (endSeed < startSeed)
80     {
81         ExecutableSupport::PrintError("Bad start & end seeds (arguments,
82             ↳ 5, 6). endSeed must not be < startSeed");
83         sane = 0;
84     }
85
86     if (endGeneration <= 0)
87     {
88         ExecutableSupport::PrintError("Bad endGeneration (argument 7).
89             ↳ endGeneration must be > 0");
90         sane = 0;
91     }
92
93     if (phase3Generation < phase2Generation)
94     {
95         ExecutableSupport::PrintError(
96             "Bad phase2Generation or phase3Generation (arguments 8,
97             ↳ 9). phase3Generation must be > phase2Generation. Both
98             ↳ must be >0");
99         sane = 0;
100    }
101
102    if (pAtoh7 < 0 || pAtoh7 > 1)
```

```

98     {
99         ExecutableSupport::PrintError("Bad pAtoh7 (argument 10). Must be
100            ↵ 0-1");
100        sane = 0;
101    }
102
103    if (pPtf1a < 0 || pPtf1a > 1)
104    {
105        ExecutableSupport::PrintError("Bad pPtf1a (argument 11). Must be
106            ↵ 0-1");
106        sane = 0;
107    }
108
109    if (png < 0 || png > 1)
110    {
111        ExecutableSupport::PrintError("Bad png (argument 12). Must be
112            ↵ 0-1");
112        sane = 0;
113    }
114
115    if (sane == 0)
116    {
117        ExecutableSupport::PrintError("Exiting with bad arguments. See
118            ↵ errors for details");
118        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
119        return exit_code;
119    }
120
121
122 /*****
123 * SIMULATOR OUTPUT SETUP
124 *****/
125
126 //Set up singletonLogFile
127 LogFile* p_log = LogFile::Instance();
128 p_log->Set(0, directoryString, filenameString);
129
130 ExecutableSupport::Print("Simulator writing file " + filenameString +
131             ↵ " to directory " + directoryString);
131
132 //Log entry counter
133     unsigned entry_number = 1;
134
135 //Write appropriate headers to log

```

```

136     if (outputMode == 0) *p_log << "Entry\tSeed\tCount\n";
137     if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic
138         Mode (0=PP;1=PD;2=DD)\n";
139     if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";
140
141 //Instance RNG
142     RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();
143
144 //Initialise pointers to relevant singleton ProliferativeTypes and
145 //Properties
146     MAKE_PTR(WildTypeCellMutationState, p_state);
147     MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
148     MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
149     MAKE_PTR(RetinalGanglion, p_RGC_fate);
150     MAKE_PTR(AmacrineHorizontal, p_AC_HC_fate);
151     MAKE_PTR(ReceptorBipolar, p_PR_BC_fate);
152     MAKE_PTR(CellLabel, p_label);
153
154 //*****
155 // SIMULATOR SETUP & RUN
156 //*****
157
158 //iterate through supplied seed range, executing one simulation per seed
159 for (unsigned seed = startSeed; seed <= endSeed; seed++)
160 {
161     if (outputMode == 2) *p_log << entry_number << "\t" << seed <<
162         "\t"; //write seed to log - sequence written by
163         cellcyclemodel objects
164
165     //initialise pointer to debugWriter
166     ColumnDataWriter* debugWriter;
167
168     //initialise SimulationTime (permits cellcyclemodel setup)
169     SimulationTime::Instance()→SetStartTime(0.0);
170
171     //Reseed the RNG with the required seed
172     p_RNG→Reseed(seed);
173
174     //Initialise a HeCellCycleModel and set it up with appropriate
175     // TiL values
176     BoijeCellCycleModel* p_cycle_model = new BoijeCellCycleModel;
177
178     if (debugOutput)

```

```

174 {
175     //Pass ColumnDataWriter to cell cycle model for debug output
176     boost::shared_ptr<ColumnDataWriter> p_debugWriter(
177         new ColumnDataWriter(directoryString, filenameString
178             ↵ + "DEBUG_" + std::to_string(seed), false, 10));
179     p_cycle_model->EnableModelDebugOutput(p_debugWriter);
180     debugWriter = &p_debugWriter;
181 }
182
183 //Setup lineages' cycle model with appropriate parameters
184 p_cycle_model->SetDimension(2);
185 p_cycle_model->SetPostMitoticType(p_PostMitotic);
186
187 //Setup vector containing lineage founder with the properly set
188 //up cell cycle model
189 std::vector<CellPtr> cells;
190 CellPtr p_cell(new Cell(p_state, p_cycle_model));
191 p_cell->SetCellProliferativeType(p_Mitotic);
192 p_cycle_model->SetModelParameters(phase2Generation,
193     ↵ phase3Generation, pAtoh7, pPtf1a, png);
194 p_cycle_model->SetSpecifiedTypes(p_RGC_fate, p_AC_HC_fate,
195     ↵ p_PR_BC_fate);
196 if (outputMode == 2)
197     ↵ p_cycle_model->EnableSequenceSampler(p_label);
198 if (outputMode == 2) p_cell->AddCellProperty(p_label);
199 p_cell->InitialiseCellCycleModel();
200 cells.push_back(p_cell);
201
202
203 //Generate 1x1 mesh for single-cell colony
204 HoneycombMeshGenerator generator(1, 1);
205 MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
206 NodesOnlyMesh<2> mesh;
207 mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);
208
209 //Setup cell population
210 NodeBasedCellPopulation<2>* cell_population(new
211     ↵ NodeBasedCellPopulation<2>(mesh, cells));
212
213 //Setup simulator & run simulation
214 boost::shared_ptr<OffLatticeSimulationPropertyStop<2>>
215     ↵ p_simulator(
216         new
217             ↵ OffLatticeSimulationPropertyStop<2>(*cell_population));

```

```

209     p_simulator→SetStopProperty(p_Mitotic); //simulation to stop if
210     ↵ no mitotic cells are left
211     p_simulator→SetDt(0.25);
212     p_simulator→SetEndTime(endGeneration);
213     p_simulator→SetOutputDirectory("UnusedSimOutput" +
214     ↵ filenameString); //unused output
215     p_simulator→Solve();
216
217
218     //Count lineage size
219     unsigned count = cell_population→GetNumRealCells();
220
221     if (outputMode == 0) *p_log << entry_number << "\t" << seed <<
222     ↵ "\t" << count << "\n";
223     if (outputMode == 2) *p_log << "\n";
224
225     //Reset for next simulation
226     SimulationTime::Destroy();
227     delete cell_population;
228     entry_number++;
229
230     if (debugOutput)
231     {
232         debugWriter→Close();
233     }
234
235
236     return exit_code;
237 }
238 ;

```

15.1.2 /apps/src/GomesSimulator.cpp

```

1 #include <iostream>
2 #include <string>
3
4 #include <cxxtest/TestSuite.h>
5 #include "ExecutableSupport.hpp"
6 #include "Exception.hpp"

```

```

7 #include "PetscTools.hpp"
8 #include "PetscException.hpp"
9
10 #include "GomesCellCycleModel.hpp"
11 #include "OffLatticeSimulationPropertyStop.hpp"
12
13 #include "AbstractCellBasedTestSuite.hpp"
14
15 #include "WildTypeCellMutationState.hpp"
16 #include "TransitCellProliferativeType.hpp"
17 #include "DifferentiatedCellProliferativeType.hpp"
18 #include "GomesRetinalNeuralFates.hpp"
19
20 #include "CellsGenerator.hpp"
21 #include "HoneycombMeshGenerator.hpp"
22 #include "NodesOnlyMesh.hpp"
23 #include "NodeBasedCellPopulation.hpp"
24 #include "VertexBasedCellPopulation.hpp"
25
26 #include "ColumnDataWriter.hpp"
27
28 int main(int argc, char *argv[])
29 {
30     ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
31     //main() returns code indicating sim run success or failure mode
32     int exit_code = ExecutableSupport::EXIT_OK;
33
34     if (argc != 15)
35     {
36         ExecutableSupport::PrintError(
37             "Wrong arguments for simulator.\nUsage (replace <> with
38             → values, pass bools as 0 or 1):\n GomesSimulator
39             → <directoryString> <filenameString>
40             → <outputModeUnsigned(0=counts,1=events,2=sequence)>
41             → <debugOutputBool> <startSeedUnsigned>
42             → <endSeedUnsigned> <endTimeDoubleHours>
43             → <cellCycleNormalMeanDouble>
44             → <cellCycleNormalStdDouble> <pPPDouble(0-1)>
45             → <pPDDouble(0-1)> <pBCDouble(0-1)> <pACDouble(0-1)>
46             → <pMGDouble(0-1)>",
47             true);
48         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
49     }
50     return exit_code;

```

```

41    }
42
43    /*****
44     * SIMULATOR PARAMETERS
45     *****/
46    std::string directoryString, filenameString;
47    int outputMode; //0 = counts; 1 = mitotic mode events; 2 = mitotic
48    ← mode sequence sampling
49    bool debugOutput;
50    unsigned startSeed, endSeed;
51    double endTime;
52    double normalMu, normalSigma, pPP, pPD, pBC, pAC, pMG; //stochastic
53    ← model parameters
54
55    //PARSE ARGUMENTS
56    directoryString = argv[1];
57    filenameString = argv[2];
58    outputMode = std::stoi(argv[3]);
59    debugOutput = std::stoul(argv[4]);
60    startSeed = std::stoul(argv[5]);
61    endSeed = std::stoul(argv[6]);
62    endTime = std::stod(argv[7]);
63    normalMu = std::stod(argv[8]);
64    normalSigma = std::stod(argv[9]);
65    pPP = std::stod(argv[10]);
66    pPD = std::stod(argv[11]);
67    pBC = std::stod(argv[12]);
68    pAC = std::stod(argv[13]);
69    pMG = std::stod(argv[14]);
70
71    /*****
72     * PARAMETER/ARGUMENT SANITY CHECK
73     *****/
74    bool sane = 1;
75
76    if (outputMode != 0 && outputMode != 1 && outputMode != 2)
77    {
78        ExecutableSupport::PrintError(
79            "Bad outputMode (argument 3). Must be 0 (counts) 1
80            ← (mitotic events) or 2 (sequence sampling)");
81        sane = 0;
82    }

```

```

81     if (endSeed < startSeed)
82     {
83         ExecutableSupport::PrintError("Bad start & end seeds (arguments,
84             ↵ 5, 6). endSeed must not be < startSeed");
85         sane = 0;
86     }
87
88     if (endTime ≤ 0)
89     {
90         ExecutableSupport::PrintError("Bad endTime (argument 7). endTime
91             ↵ must be > 0");
92         sane = 0;
93     }
94
95     if (normalMu ≤ 0 || normalSigma ≤ 0)
96     {
97         ExecutableSupport::PrintError("Bad cell cycle normal mean or std
98             ↵ (arguments 8, 9). Must be >0");
99         sane = 0;
100    }
101
102    ExecutableSupport::PrintError(
103        "Bad mitotic mode probabilities (arguments 10, 11). pPP +
104            ↵ pPD should be ≥0, ≤1, sum should not exceed 1");
105    sane = 0;
106
107    if (pBC + pAC + pMG > 1 || pBC > 1 || pBC < 0 || pAC > 1 ||
108        ↵ pAC < 0 || pMG > 1 || pMG < 0)
109    {
110        ExecutableSupport::PrintError(
111            "Bad specification probabilities (arguments 12, 13, 14).
112                ↵ pBC, pAC, pMG should be ≥0, ≤1, sum should not
113                ↵ exceed 1");
114        sane = 0;
115    }
116
117    if (sane == 0)
118    {
119        ExecutableSupport::PrintError("Exiting with bad arguments. See
120            ↵ errors for details");

```

```

116     exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
117     return exit_code;
118 }
119
120 /*****
121 * SIMULATOR OUTPUT SETUP
122 *****/
123
124 //Set up singleton LogFile
125 LogFile* p_log = LogFile::Instance();
126 p_log->Set(0, directoryString, filenameString);
127
128 ExecutableSupport::Print("Simulator writing file " + filenameString +
129   " to directory " + directoryString);
130
131 //Log entry counter
132 unsigned entry_number = 1;
133
134 //Write appropriate headers to log
135 if (outputMode == 0) *p_log << "Entry\tSeed\tCount\n";
136 if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic
137   Mode (0=PP;1=PD;2=DD)\n";
138 if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";
139
140 //Instance RNG
141 RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();
142
143 //Initialise pointers to relevant singleton ProliferativeTypes and
144   Properties
145 MAKE_PTR(WildTypeCellMutationState, p_state);
146 MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
147 MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
148 MAKE_PTR(RodPhotoreceptor, p_RPh_fate);
149 MAKE_PTR(AmacrineCell, p_AC_fate);
150 MAKE_PTR(BipolarCell, p_BC_fate);
151 MAKE_PTR(MullerGlia, p_MG_fate);
152 MAKE_PTR(CellLabel, p_label);
153
154 /*****
155 * SIMULATOR SETUP & RUN
156 *****/
157
158 //iterate through supplied seed range, executing one simulation per seed

```

```

156     for (unsigned seed = startSeed; seed <= endSeed; seed++)
157     {
158         if (outputMode == 2) *p_log << entry_number << "\t" << seed <<
159             "\t"; //write seed to log - sequence written by
160             // cellcyclemodel objects
161
162         //initialise pointer to debugWriter
163         ColumnDataWriter* debugWriter;
164
165         //initialise SimulationTime (permits cellcyclemodel setup)
166         SimulationTime::Instance()→SetStartTime(0.0);
167
168         //Reseed the RNG with the required seed
169         p_RNG→Reseed(seed);
170
171         //Initialise a HeCellCycleModel and set it up with appropriate
172             // TiL values
173         GomesCellCycleModel* p_cycle_model = new GomesCellCycleModel;
174
175         if (debugOutput)
176         {
177             //Pass ColumnDataWriter to cell cycle model for debug output
178             boost::shared_ptr<ColumnDataWriter> p_debugWriter(
179                 new ColumnDataWriter(directoryString, filenameString
180                     + "DEBUG_" + std::to_string(seed), false, 10));
181             p_cycle_model→EnableModelDebugOutput(p_debugWriter);
182             debugWriter = &p_debugWriter;
183         }
184
185         //Setup lineages' cycle model with appropriate parameters
186         p_cycle_model→SetDimension(2);
187         p_cycle_model→SetPostMitoticType(p_PostMitotic);
188
189         //Setup vector containing lineage founder with the properly set
190             // up cell cycle model
191         std::vector<CellPtr> cells;
192         CellPtr p_cell(new Cell(p_state, p_cycle_model));
193         p_cell→SetCellProliferativeType(p_Mitotic);
194         p_cycle_model→SetModelParameters(normalMu, normalSigma, pPP,
195             pPD, pBC, pAC, pMG);
196         p_cycle_model→SetModelProperties(p_RPh_fate, p_AC_fate,
197             p_BC_fate, p_MG_fate);

```

```

191     if (outputMode == 2)
192         ↵ p_cycle_model->EnableSequenceSampler(p_label);
193     if (outputMode == 2) p_cell->AddCellProperty(p_label);
194     p_cell->InitialiseCellCycleModel();
195     cells.push_back(p_cell);

196     //Generate 1x1 mesh for single-cell colony
197     HoneycombMeshGenerator generator(1, 1);
198     MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
199     NodesOnlyMesh<2> mesh;
200     mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);

201     //Setup cell population
202     NodeBasedCellPopulation<2>* cell_population(new
203         ↵ NodeBasedCellPopulation<2>(mesh, cells));

204     //Setup simulator & run simulation
205     boost::shared_ptr<OffLatticeSimulationPropertyStop<2>>
206         ↵ p_simulator(
207             new
208                 ↵ OffLatticeSimulationPropertyStop<2>(*cell_population));
209     p_simulator->SetStopProperty(p_Mitotic); //simulation to stop if
210     ↵ no mitotic cells are left
211     p_simulator->SetDt(0.25);
212     p_simulator->SetEndTime(endTime);
213     p_simulator->SetOutputDirectory("UnusedSimOutput" +
214         ↵ filenameString); //unused output
215     p_simulator->Solve();

216     //Count lineage size
217     unsigned count = cell_population->GetNumRealCells();

218     if (outputMode == 0) *p_log << entry_number << "\t" << seed <<
219         ↵ "\t" << count << "\n";
220     if (outputMode == 2) *p_log << "\n";

221     //Reset for next simulation
222     SimulationTime::Destroy();
223     delete cell_population;
224     entry_number++;

225     if (debugOutput)
226     {

```

```

227         debugWriter->Close();
228     }
229
230 }
231
232 p_RNG->Destroy();
233 LogFile::Close();
234
235 return exit_code;
236 }
237 ;

```

15.1.3 /apps/src/HeSimulator.cpp

```

1 #include <iostream>
2 #include <string>
3
4 #include <cxxtest/TestSuite.h>
5 #include "ExecutableSupport.hpp"
6 #include "Exception.hpp"
7 #include "PetscTools.hpp"
8 #include "PetscException.hpp"
9
10 #include "HeCellCycleModel.hpp"
11 #include "OffLatticeSimulationPropertyStop.hpp"
12
13 #include "AbstractCellBasedTestSuite.hpp"
14
15 #include "WildTypeCellMutationState.hpp"
16 #include "TransitCellProliferativeType.hpp"
17 #include "DifferentiatedCellProliferativeType.hpp"
18
19 #include "CellsGenerator.hpp"
20 #include "HoneycombMeshGenerator.hpp"
21 #include "NodesOnlyMesh.hpp"
22 #include "NodeBasedCellPopulation.hpp"
23 #include "VertexBasedCellPopulation.hpp"
24
25 #include "ColumnDataWriter.hpp"
26
27 int main(int argc, char *argv[])
28 {

```

```

29     ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
30     //main() returns code indicating sim run success or failure mode
31     int exit_code = ExecutableSupport::EXIT_OK;
32
33     if (argc != 22 && argc != 20)
34     {
35         ExecutableSupport::PrintError(
36             "Wrong arguments for simulator.\nUsage (replace< with
37             → values, pass bools as 0 or 1):\nStochastic
38             → Mode:\nHeSimulator <directoryString> <filenameString>
39             → <outputModeUnsigned(0=counts,1=events,2=sequence)>
40             → <deterministicBool=0>
41             → <fixtureUnsigned(0=He;1=Wan;2=test)>
42             → <founderAth5Mutant?Bool> <debugOutputBool>
43             → <startSeedUnsigned> <endSeedUnsigned>
44             → <inductionTimeDoubleHours>
45             → <earliestLineageStartDoubleHours>
46             → <latestLineageStartDoubleHours> <endTimeDoubleHours>
47             → <mMitoticModePhase2Double> <mMitoticModePhase3Double>
48             → <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)>
49             → <pPD1Double(0-1)> <pPP1Double(0-1)>
50             → <pPD1Double(0-1)>\nDeterministic Mode:\nHeSimulator
51             → <directoryString> <filenameString>
52             → <outputModeUnsigned(0=counts,1=events,2=sequence)>
53             → <deterministicBool=1>
54             → <fixtureUnsigned(0=He;1=Wan;2=test)>
55             → <founderAth5Mutant?Bool> <debugOutputBool>
56             → <startSeedUnsigned> <endSeedUnsigned>
57             → <inductionTimeDoubleHours>
58             → <earliestLineageStartDoubleHours>
59             → <latestLineageStartDoubleHours> <endTimeDoubleHours>
60             → <phase1ShapeDouble(>0)> <phase1ScaleDouble(>0)>
61             → <phase2ShapeDouble(>0)> <phase2ScaleDouble(>0)>
62             → <phaseBoundarySisterShiftWidthDouble>\n",
63             true);
64         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
65         return exit_code;
66     }
67
68     /*****
69      * SIMULATOR PARAMETERS
70      *****/
71
72     std::string directoryString, filenameString;

```

```

46     int outputMode; //0 = counts; 1 = mitotic mode events; 2 = mitotic
47     ↵ mode sequence sampling
48     bool deterministicMode, ath5founder, debugOutput;
49     unsigned fixture, startSeed, endSeed; //fixture 0 = He2012; 1 =
50     ↵ Wan2016
51     double inductionTime, earliestLineageStartTime,
52     ↵ latestLineageStartTime, endTime;
53     double mitoticModePhase2, mitoticModePhase3, pPP1, pPD1, pPP2, pPD2,
54     ↵ pPP3, pPD3; //stochastic model parameters
55     double phase1Shape, phase1Scale, phase2Shape, phase2Scale,
56     ↵ phaseSisterShiftWidth, phaseOffset;

57     //PARSE ARGUMENTS
58     directoryString = argv[1];
59     filenameString = argv[2];
60     outputMode = std::stoi(argv[3]);
61     deterministicMode = std::stoul(argv[4]);
62     fixture = std::stoul(argv[5]);
63     ath5founder = std::stoul(argv[6]);
64     debugOutput = std::stoul(argv[7]);
65     startSeed = std::stoul(argv[8]);
66     endSeed = std::stoul(argv[9]);
67     inductionTime = std::stod(argv[10]);
68     earliestLineageStartTime = std::stod(argv[11]);
69     latestLineageStartTime = std::stod(argv[12]);
70     endTime = std::stod(argv[13]);

71     if (deterministicMode == 0)
72     {
73         mitoticModePhase2 = std::stod(argv[14]);
74         mitoticModePhase3 = std::stod(argv[15]);
75         pPP1 = std::stod(argv[16]);
76         pPD1 = std::stod(argv[17]);
77         pPP2 = std::stod(argv[18]);
78         pPD2 = std::stod(argv[19]);
79         pPP3 = std::stod(argv[20]);
80         pPD3 = std::stod(argv[21]);
81     }
82     else if (deterministicMode == 1)
83     {
84         phase1Shape = std::stod(argv[14]);
85         phase1Scale = std::stod(argv[15]);
86         phase2Shape = std::stod(argv[16]);

```

```

84     phase2Scale = std::stod(argv[17]);
85     phaseSisterShiftWidth = std::stod(argv[18]);
86     phaseOffset = std::stod(argv[19]);
87 }
88 else
89 {
90     ExecutableSupport::PrintError("Bad deterministicMode (argument
91     ↵ 4). Must be 0 or 1");
92     exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
93     return exit_code;
94 }
95
96 /******
97 * PARAMETER/ARGUMENT SANITY CHECK
98 *****/
99 bool sane = 1;
100
101 if (outputMode != 0 && outputMode != 1 && outputMode != 2)
102 {
103     ExecutableSupport::PrintError(
104         "Bad outputMode (argument 3). Must be 0 (counts) 1
105         ↵ (mitotic events) or 2 (sequence sampling)");
106     sane = 0;
107 }
108
109 if (fixture != 0 && fixture != 1 && fixture != 2)
110 {
111     ExecutableSupport::PrintError("Bad fixture (argument 5). Must be
112         ↵ 0 (He), 1 (Wan), or 2 (validation/test)");
113     sane = 0;
114 }
115
116 if (ath5founder != 0 && ath5founder != 1)
117 {
118     ExecutableSupport::PrintError("Bad ath5founder (argument 6). Must
119         ↵ be 0 (wild type) or 1 (ath5 mutant)");
120     sane = 0;
121 }
122
123 if (endSeed < startSeed)
124 {
125     ExecutableSupport::PrintError("Bad start & end seeds (arguments,
126         ↵ 8, 9). endSeed must not be < startSeed");

```

```
122     sane = 0;
123 }
124
125 if (inductionTime >= endTime)
126 {
127     ExecutableSupport::PrintError("Bad latestLineageStartTime
128         ↪ (argument 10). Must be <endTime(arg13)");
129     sane = 0;
130 }
131 if (earliestLineageStartTime >= endTime || earliestLineageStartTime
132     ↪ >= latestLineageStartTime)
133 {
134     ExecutableSupport::PrintError(
135         "Bad earliestLineageStartTime (argument 11). Must be
136             ↪ <endTime(arg13), <latestLineageStarTime (arg12)");
137     sane = 0;
138 }
139 if (latestLineageStartTime > endTime)
140 {
141     ExecutableSupport::PrintError("Bad latestLineageStartTime
142         ↪ (argument 12). Must be <=endTime(arg13)");
143     sane = 0;
144 }
145
146 if (deterministicMode == 0)
147 {
148     if (mitoticModePhase2 < 0)
149     {
150         ExecutableSupport::PrintError("Bad mitoticModePhase2
151             ↪ (argument 14). Must be >0");
152         sane = 0;
153     }
154     if (mitoticModePhase3 < 0)
155     {
156         ExecutableSupport::PrintError("Bad mitoticModePhase3
157             ↪ (argument 15). Must be >0");
158         sane = 0;
159     }
160     if (pPP1 + pPD1 > 1 || pPP1 > 1 || pPP1 < 0 || pPD1 > 1 || pPD1 <
161         ↪ 0)
162     {
163         ExecutableSupport::PrintError(
```

```

157             "Bad phase 1 probabilities (arguments 16, 17). pPP1 +
158             ↳ pPD1 should be  $\geq 0$ ,  $\leq 1$ , sum should not exceed
159             ↳ 1");
160             sane = 0;
161         }
162         if (pPP2 + pPD2 > 1 || pPP2 > 1 || pPP2 < 0 || pPD2 > 1 || pPD2 <
163             ↳ 0)
164         {
165             ExecutableSupport::PrintError(
166                 "Bad phase 2 probabilities (arguments 18, 19). pPP2 +
167                 ↳ pPD2 should be  $\geq 0$ ,  $\leq 1$ , sum should not exceed
168                 ↳ 1");
169                 sane = 0;
170             }
171         }
172     }
173
174     if (deterministicMode == 1)
175     {
176         if (phase1Shape  $\leq 0$ )
177         {
178             ExecutableSupport::PrintError("Bad phase1Shape (argument 14).
179             ↳ Must be  $>0$ ");
180             sane = 0;
181         }
182         if (phase1Scale  $\leq 0$ )
183         {
184             ExecutableSupport::PrintError("Bad phase1Scale (argument 15).
185             ↳ Must be  $>0$ ");
186             sane = 0;
187         }
188         if (phase2Shape  $\leq 0$ )
189         {
190             ExecutableSupport::PrintError("Bad phase1Shape (argument 16).
191             ↳ Must be  $>0$ ");

```

```

189         sane = 0;
190     }
191     if (phase2Scale <= 0)
192     {
193         ExecutableSupport::PrintError("Bad phase1Scale (argument 17).
194             ↪ Must be >0");
195         sane = 0;
196     }
197     if (phaseSisterShiftWidth <= 0)
198     {
199         ExecutableSupport::PrintError("Bad phaseSisterShiftWidth
200             ↪ (argument 18). Must be >0");
201         sane = 0;
202     }
203     if (sane == 0)
204     {
205         ExecutableSupport::PrintError("Exiting with bad arguments. See
206             ↪ errors for details");
207         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
208         return exit_code;
209     }
210
211 /*****
212 * SIMULATOR OUTPUT SETUP
213 *****/
214
215 //Set up singleton LogFile
216    LogFile* p_log = LogFile::Instance();
217     p_log->Set(0, directoryString, filenameString);
218
219     ExecutableSupport::Print("Simulator writing file " + filenameString +
220             ↪ " to directory " + directoryString);
221
222 //Log entry counter
223     unsigned entry_number = 1;
224
225 //Write appropriate headers to log
226     if (outputMode == 0) *p_log << "Entry\tInduction Time
227             ↪ (h)\tSeed\tCount\n";

```

```

226     if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic
227         Mode (0=PP;1=PD;2=DD)\n";
228     if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";
229
230 //Instance RNG
231     RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();
232
233 //Initialise pointers to relevant singleton ProliferativeTypes and
234     ↪ Properties
235     MAKE_PTR(WildTypeCellMutationState, p_state);
236     MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
237     MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
238     MAKE_PTR(Ath5Mo, p_Morpholino);
239     MAKE_PTR(CellLabel, p_label);
240
241 /******
242     * SIMULATOR SETUP & RUN
243     *****/
244
245 //iterate through supplied seed range, executing one simulation per seed
246     for (unsigned seed = startSeed; seed <= endSeed; seed++)
247     {
248         if (outputMode == 2) *p_log << entry_number << "\t" << seed <<
249             "\t"; //write seed to log - sequence written by
250             ↪ cellcyclemodel objects
251
252         //initialise pointer to debugWriter
253         ColumnDataWriter* debugWriter;
254
255         //initialise SimulationTime (permits cellcyclemodel setup)
256         SimulationTime::Instance()→SetStartTime(0.0);
257
258         //Reseed the RNG with the required seed
259         p_RNG→Reseed(seed);
260
261         //Initialise a HeCellCycleModel and set it up with appropriate
262             ↪ TiL values
263         HeCellCycleModel* p_cycle_model = new HeCellCycleModel;
264
265         if (debugOutput)
266         {
267             //Pass ColumnDataWriter to cell cycle model for debug output
268             boost::shared_ptr<ColumnDataWriter> p_debugWriter(

```

```

264             new ColumnDataWriter(directoryString, filenameString
265                 ↵ + "DEBUG_" + std::to_string(seed), false, 10));
266         p_cycle_model→EnableModelDebugOutput(p_debugWriter);
267         debugWriter = &p_debugWriter;
268     }
269
270     double currTIL; //Time in Lineage offset for lineages induced
271     ↵ after first mitosis
272     double lineageStartTime; //first mitosis time (hp)
273     double currSimEndTime; //simulation end time (hp);
274
275
276     ↵ ****
277 * Time in Lineage Generation Fixtures & Cell Cycle Model Setup
278
279     ↵ ****
280
281     if (fixture == 0) //He 2012-type fixture - even distribution
282     ↵ across nasal-temporal axis
283     {
284         //generate lineage start time from even random distro across
285         ↵ earliest-latest start time figures
286         lineageStartTime = (p_RNG→ranf() * (latestLineageStartTime -
287             ↵ earliestLineageStartTime))
288             + earliestLineageStartTime;
289         //this reflects induction of cells after the lineages' first
290         ↵ mitosis
291         if (lineageStartTime < inductionTime)
292         {
293             currTIL = inductionTime - lineageStartTime;
294             currSimEndTime = endTime - inductionTime;
295             if (outputMode == 1)
296                 ↵ p_cycle_model→EnableModeEventOutput(inductionTime,
297                     ↵ seed);
298         }
299         //if the lineage starts after the induction time, give it
300         ↵ zero & TIL run the appropriate-length simulation
301         // (ie. the endTime is reduced by the amount of time after
302             ↵ induction that the first mitosis occurs)
303         if (lineageStartTime ≥ inductionTime)
304         {
305             currTIL = 0.0;
306             currSimEndTime = endTime - lineageStartTime;
307         }
308     }
309
310     ↵ ****

```

```

295         if (outputMode == 1)
296             → p_cycle_model→EnableModeEventOutput(lineageStartTime,
297             → seed);
298     }
299
300     else if (fixture == 1) //Wan 2016-type fixture - each lineage
301         → founder selected randomly across residency time, simulator
302         → allowed to run until end of residency time
303         //passing residency time (as latestLineageStartTime) and endTime
304         → separately allows for creation of "shadow CMZ" population
305         //this allows investigation of different assumptions about how
306         → Wan et al.'s model output was generated
307     {
308         //generate random lineage start time from even random distro
309         → across CMZ residency time
310         currTiL = p_RNG→ranf() * latestLineageStartTime;
311         currSimEndTime = std::max(.05, endTime - currTiL); //minimum
312         → 1 timestep, prevents 0 timestep SimulationTime error
313         if (outputMode == 1) p_cycle_model→EnableModeEventOutput(0,
314             → seed);
315     }
316     else if (fixture == 2) //validation fixture- all founders have
317         → TiL given by induction time
318     {
319         currTiL = inductionTime;
320         currSimEndTime = endTime;
321     }
322
323     //Setup lineages' cycle model with appropriate parameters
324     p_cycle_model→SetDimension(2);
325     //p_cycle_model→SetPostMitoticType(p_PostMitotic);

326     if (!deterministicMode)
327     {
328         p_cycle_model→SetModelParameters(currTiL, mitoticModePhase2,
329             → mitoticModePhase2 + mitoticModePhase3, pPP1,
330                 → pPD1, pPP2, pPD2, pPP3,
331                     → pPD3);
332     }
333     else
334     {
335         //Gamma-distribute phase3 boundary

```

```

326         double currPhase2Boundary = phaseOffset +
327             ↵ p_RNG->GammaRandomDeviate(phase1Shape, phase1Scale);
328         double currPhase3Boundary = currPhase2Boundary +
329             ↵ p_RNG->GammaRandomDeviate(phase2Shape, phase2Scale);
330     }
331
332     if (outputMode == 2) p_cycle_model->EnableSequenceSampler();
333
334     //Setup vector containing lineage founder with the properly set
335     ↵ up cell cycle model
336     std::vector<CellPtr> cells;
337     CellPtr p_cell(new Cell(p_state, p_cycle_model));
338     p_cell->SetCellProliferativeType(p_Mitotic);
339     if (ath5founder == 1) p_cell->AddCellProperty(p_Morpholino);
340     if (outputMode == 2) p_cell->AddCellProperty(p_label);
341     p_cell->InitialiseCellCycleModel();
342     cells.push_back(p_cell);
343
344     //Generate 1x1 mesh for single-cell colony
345     HoneycombMeshGenerator generator(1, 1);
346     MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
347     NodesOnlyMesh<2> mesh;
348     mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);
349
350     //Setup cell population
351     NodeBasedCellPopulation<2>* cell_population(new
352         ↵ NodeBasedCellPopulation<2>(mesh, cells));
353
354     //Setup simulator & run simulation
355     boost::shared_ptr<OffLatticeSimulationPropertyStop<2>>
356         ↵ p_simulator(
357             new
358                 ↵ OffLatticeSimulationPropertyStop<2>(*cell_population));
359     p_simulator->SetStopProperty(p_Mitotic); //simulation to stop if
360         ↵ no mitotic cells are left
361     p_simulator->SetDt(0.05);
362     p_simulator->SetEndTime(currSimEndTime);
363     p_simulator->SetOutputDirectory("UnusedSimOutput" +
364         ↵ filenameString); //unused output

```

```

359     p_simulator→Solve();
360
361     //Count lineage size
362     unsigned count = cell_population→GetNumRealCells();
363
364     if (outputMode == 0) *p_log << entry_number << "\t" <<
365         → inductionTime << "\t" << seed << "\t" << count << "\n";
366     if (outputMode == 2) *p_log << "\n";
367
368     //Reset for next simulation
369     SimulationTime::Destroy();
370     delete cell_population;
371     entry_number++;
372
373     if (debugOutput)
374     {
375         debugWriter→Close();
376     }
377
378
379     p_RNG→Destroy();
380     LogFile::Close();
381
382     return exit_code;
383 }
384 ;

```

15.1.4 /apps/src/WanSimDebug.cpp

```

1 #include <iostream>
2 #include <string>
3
4 #include <cxxtest/TestSuite.h>
5 #include "ExecutableSupport.hpp"
6 #include "Exception.hpp"
7 #include "PetscTools.hpp"
8 #include "PetscException.hpp"
9
10 #include "WanStemCellCycleModel.hpp"
11 #include "HeCellCycleModel.hpp"
12 #include "OffLatticeSimulation.hpp"

```

```
13
14 #include "AbstractCellBasedTestSuite.hpp"
15
16 #include "WildTypeCellMutationState.hpp"
17 #include "StemCellProliferativeType.hpp"
18 #include "TransitCellProliferativeType.hpp"
19 #include "DifferentiatedCellProliferativeType.hpp"
20
21 #include "CellsGenerator.hpp"
22 #include "HoneycombMeshGenerator.hpp"
23 #include "NodesOnlyMesh.hpp"
24 #include "NodeBasedCellPopulation.hpp"
25 #include "VertexBasedCellPopulation.hpp"
26
27 #include "CellProliferativeTypesCountWriter.hpp"
28
29
30 int main(int argc, char *argv[])
31 {
32     ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
33     //main() returns code indicating sim run success or failure mode
34     int exit_code = ExecutableSupport::EXIT_OK;
35
36     /*****
37     * SIMULATOR PARAMETERS
38     *****/
39     std::string directoryString, filenameString;
40
41     //PARSE ARGUMENTS
42     directoryString = argv[1];
43     filenameString = argv[2];
44
45     /*****
46     * SIMULATOR OUTPUT SETUP
47     *****/
48
49 //Set up singleton LogFile
50    LogFile* p_log = LogFile::Instance();
51     p_log->Set(0, directoryString, filenameString);
52
53     ExecutableSupport::Print("Simulator writing file " + filenameString +
54     " to directory " + directoryString);
```

```

55 //Log entry counter
56     //unsigned entry_number = 1;
57
58 //Write appropriate header to log
59     *p_log <<
60         "Entry\tTotalCells\tStemCount\tProgenitorCount\tPostMitoticCount\n";
61
62 //Instance RNG
63     RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();
64
65 //Initialise pointers to relevant singleton ProliferativeTypes and
66     Properties
67     boost::shared_ptr<AbstractCellProperty>
68         p_state(CellPropertyRegistry :: Instance()→Get<WildTypeCellMutationState>());
69     boost::shared_ptr<AbstractCellProperty>
70         p_Stem(CellPropertyRegistry :: Instance()→Get<StemCellProliferativeType>());
71     boost::shared_ptr<AbstractCellProperty>
72         p_Transit(CellPropertyRegistry :: Instance()→Get<TransitCellProliferativeType>());
73     boost::shared_ptr<AbstractCellProperty>
74         p_PostMitotic(CellPropertyRegistry :: Instance()→Get<DifferentiatedCellProlife
75
76     /*****
77     * SIMULATOR SETUP & RUN
78     *****/
79
80     //initialise SimulationTime (permits cellcyclemodel setup)
81     SimulationTime::Instance()→SetStartTime(0.0);
82
83     std::vector<CellPtr> cells;
84
85     WanStemCellCycleModel* p_stem_model = new WanStemCellCycleModel;
86
87     boost::shared_ptr<ColumnDataWriter> p_debugWriter(
88         new ColumnDataWriter(directoryString, filenameString +
89             "DEBUG_WAN", false, 10));
90
91     p_stem_model→SetDimension(2);
92     p_stem_model→EnableModelDebugOutput(p_debugWriter);
93     CellPtr p_cell(new Cell(p_state, p_stem_model));
94     p_cell→InitialiseCellCycleModel();
95     cells.push_back(p_cell);

```

```

91     //Generate 1x#cells mesh for abstract colony
92     HoneycombMeshGenerator generator(1, 1);
93     MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
94     NodesOnlyMesh<2> mesh;
95     mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);

96
97     //Setup cell population
98     boost::shared_ptr<NodeBasedCellPopulation<2>> cell_population(new
99         → NodeBasedCellPopulation<2>(mesh, cells));
100
101    → cell_population→AddCellPopulationCountWriter<CellProliferativeTypesCountWriter>(p_stem_model);
102
103    //Setup simulator & run simulation
104    boost::shared_ptr<OffLatticeSimulation<2>> p_simulator(
105        new OffLatticeSimulation<2>(*cell_population));
106    p_simulator→SetDt(1);
107    p_simulator→SetOutputDirectory(directoryString + "/WanDebug");
108    p_simulator→SetEndTime(100);
109    p_simulator→Solve();

110
111    std::vector<unsigned> prolTypes =
112        → cell_population→GetCellProliferativeTypeCount();
113    unsigned realCells = cell_population→GetNumRealCells();
114    unsigned allCells = cell_population→GetNumAllCells();

115    Timer::Print("stem: " + std::to_string(prolTypes[0]) + " transit: " +
116        → std::to_string(prolTypes[1]) + " postmitotic: " +
117        → std::to_string(prolTypes[2]));
118    Timer::Print("realcells: " + std::to_string(realCells) + " allcells: " +
119        → std::to_string(allCells));

120
121     //Reset for next simulation
122     SimulationTime::Destroy();
123     cell_population.reset();

124
125     p_RNG→Destroy();
126     LogFile::Close();

127
128     return exit_code;
129 }
130 ;

```

15.1.5 /apps/src/WanSimulator.cpp

```
1 #include <iostream>
2 #include <string>
3
4 #include <cxxtest/TestSuite.h>
5 #include "ExecutableSupport.hpp"
6 #include "Exception.hpp"
7 #include "PetscTools.hpp"
8 #include "PetscException.hpp"
9
10 #include "WanStemCellCycleModel.hpp"
11 #include "HeCellCycleModel.hpp"
12 #include "OffLatticeSimulationPropertyStop.hpp"
13
14 #include "AbstractCellBasedTestSuite.hpp"
15
16 #include "WildTypeCellMutationState.hpp"
17 #include "StemCellProliferativeType.hpp"
18 #include "TransitCellProliferativeType.hpp"
19 #include "DifferentiatedCellProliferativeType.hpp"
20
21 #include "CellsGenerator.hpp"
22 #include "HoneycombMeshGenerator.hpp"
23 #include "NodesOnlyMesh.hpp"
24 #include "NodeBasedCellPopulation.hpp"
25 #include "VertexBasedCellPopulation.hpp"
26
27 #include "CellProliferativeTypesCountWriter.hpp"
28
29 int main(int argc, char *argv[])
30 {
31     ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
32     //main() returns code indicating sim run success or failure mode
33     int exit_code = ExecutableSupport::EXIT_OK;
34
35     if (argc != 23)
36     {
37         ExecutableSupport::PrintError(
```

```

38         "Wrong arguments for simulator.\nUsage (replace <> with
39         ← values, pass bools as 0 or 1):\n WanSimulator
40         ← <directoryString> <startSeedUnsigned>
41         ← <endSeedUnsigned> <cmzResidencyTimeDoubleHours>
42         ← <stemDivisorDouble>
43         ← <meanProgenitorPopualtion@3dpfDouble>
44         ← <stdProgenitorPopulation@3dpfDouble>
45         ← <stemGammaShiftDouble> <stemGammaShapeDouble>
46         ← <stemGammaScaleDouble> <progenitorGammaShiftDouble>
47         ← <progenitorGammaShapeDouble>
48         ← <progenitorGammaScaleDouble>
49         ← <progenitorSisterShiftDouble>
50         ← <mMitoticModePhase2Double> <mMitoticModePhase3Double>
51         ← <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)>
52         ← <pPD1Double(0-1)> <pPP1Double(0-1)>
53         ← <pPD1Double(0-1)>",
54         true);
55     exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
56     return exit_code;
57 }
58
59 //*****
60 * SIMULATOR PARAMETERS
61 *****/
62 std::string directoryString;
63 unsigned startSeed, endSeed;
64 double cmzResidencyTime;
65 double stemDivisor, progenitorMean, progenitorStd;
66 double stemGammaShift, stemGammaShape, stemGammaScale,
67     → progenitorGammaShift, progenitorGammaShape,
68     → progenitorGammaScale, progenitorGammaSister;
69 double mitoticModePhase2, mitoticModePhase3, pPP1, pPD1, pPP2, pPD2,
70     → pPP3, pPD3; //stochastic He model parameters
71
72 //PARSE ARGUMENTS
73 directoryString = argv[1];
74 startSeed = std::stoul(argv[2]);
75 endSeed = std::stoul(argv[3]);
76 cmzResidencyTime = std::stod(argv[4]);
77 //starting cell number distributions
78 stemDivisor = std::stod(argv[5]);
79 progenitorMean = std::stod(argv[6]);
80 progenitorStd = std::stod(argv[7]);

```

```

64    //cycle duration params
65    stemGammaShift = std::stod(argv[8]);
66    stemGammaShape = std::stod(argv[9]);
67    stemGammaScale = std::stod(argv[10]);
68    progenitorGammaShift = std::stod(argv[11]);
69    progenitorGammaShape = std::stod(argv[12]);
70    progenitorGammaScale = std::stod(argv[13]);
71    progenitorGammaSister = std::stod(argv[14]);
72    //He model params
73    mitoticModePhase2 = std::stod(argv[15]);
74    mitoticModePhase3 = std::stod(argv[16]);
75    pPP1 = std::stod(argv[17]);
76    pPD1 = std::stod(argv[18]);
77    pPP2 = std::stod(argv[19]);
78    pPD2 = std::stod(argv[20]);
79    pPP3 = std::stod(argv[21]);
80    pPD3 = std::stod(argv[22]);
81
82    std::vector<double> stemOffspringParams = { mitoticModePhase2,
83        ↪ mitoticModePhase2 + mitoticModePhase3, pPP1, pPD1,
84                                ↪ pPP2, pPD2, pPP3, pPD3,
85                                ↪ progenitorGammaShift,
86                                ↪ progenitorGammaShape,
87                                ↪ progenitorGammaScale,
88                                ↪ progenitorGammaSister
89                                ↪ };
90
91    /*****
92     * PARAMETER/ARGUMENT SANITY CHECK
93     *****/
94
95    bool sane = 1;
96
97    if (endSeed < startSeed)
98    {
99        ExecutableSupport::PrintError("Bad start & end seeds (arguments,
100            ↪ 3, 4). endSeed must not be < startSeed");
101        sane = 0;
102    }
103
104    if (cmzResidencyTime < 0)
105    {
106        ExecutableSupport::PrintError("Bad CMZ residency time (argument
107            ↪ 5). cmzResidencyTime must be positive-valued");
108    }

```

```

100         sane = 0;
101     }
102
103     if (stemDivisor <= 0)
104     {
105         ExecutableSupport::PrintError("Bad stemDivisor (argument 6).
106             ↳ stemDivisor must be positive-valued");
107         sane = 0;
108     }
109
110     if (progenitorMean <= 0 || progenitorStd <= 0)
111     {
112         ExecutableSupport::PrintError("Bad progenitorMean or
113             ↳ progenitorStd (arguments 7,8). Must be positive-valued");
114         sane = 0;
115     }
116
117     if (stemGammaShift < 0 || stemGammaShape < 0 || stemGammaScale < 0)
118     {
119         ExecutableSupport::PrintError(
120             "Bad stemGammaShift, stemGammaShape, or stemGammaScale
121                 ↳ (arguments 9, 10, 11). Shifts must be ≥0, cycle
122                 ↳ shape and scale params must be positive-valued");
123         sane = 0;
124     }
125
126     if (progenitorGammaShift < 0 || progenitorGammaShape < 0 ||
127         ↳ progenitorGammaScale < 0 || progenitorGammaSister < 0)
128     {
129         ExecutableSupport::PrintError(
130             "Bad progenitorGammaShift, progenitorGammaShape,
131                 ↳ progenitorGammaScale, or progenitorGammaSisterShift
132                 ↳ (arguments 12,13,14,15). Shifts must be ≥0, cycle
133                 ↳ shape and scale params must be positive-valued");
134         sane = 0;
135     }
136
137     if (mitoticModePhase2 < 0)
138     {
139         ExecutableSupport::PrintError("Bad mitoticModePhase2 (argument
140             ↳ 14). Must be >0");
141         sane = 0;
142     }
143

```

```

134     if (mitoticModePhase3 < 0)
135     {
136         ExecutableSupport::PrintError("Bad mitoticModePhase3 (argument
137             ↳ 15). Must be >0");
138         sane = 0;
139     }
140     if (pPP1 + pPD1 > 1 || pPP1 > 1 || pPP1 < 0 || pPD1 > 1 || pPD1 < 0)
141     {
142         ExecutableSupport::PrintError(
143             "Bad phase 1 probabilities (arguments 16, 17). pPP1 +
144                 ↳ pPD1 should be ≥0, ≤1, sum should not exceed 1");
145         sane = 0;
146     }
147     if (pPP2 + pPD2 > 1 || pPP2 > 1 || pPP2 < 0 || pPD2 > 1 || pPD2 < 0)
148     {
149         ExecutableSupport::PrintError(
150             "Bad phase 2 probabilities (arguments 18, 19). pPP2 +
151                 ↳ pPD2 should be ≥0, ≤1, sum should not exceed 1");
152         sane = 0;
153     }
154     if (pPP3 + pPD3 > 1 || pPP3 > 1 || pPP3 < 0 || pPD3 > 1 || pPD3 < 0)
155     {
156         ExecutableSupport::PrintError(
157             "Bad phase 3 probabilities (arguments 20, 21). pPP3 +
158                 ↳ pPD3 should be ≥0, ≤1, sum should not exceed 1");
159         sane = 0;
160     }
161     if (sane == 0)
162     {
163         ExecutableSupport::PrintError("Exiting with bad arguments. See
164             ↳ errors for details");
165         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
166         return exit_code;
167     }
168
169     ****
170     * SIMULATOR SETUP & RUN
171     ****
172
173     ExecutableSupport::Print("Simulator writing files to directory " +
174             ↳ directoryString);
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1146
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2488
2489
2490
2491
2492
24
```

```

171 //Instance RNG
172     RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();
173
174 //Initialise pointers to relevant singleton ProliferativeTypes and
175     ↳ Properties
176     boost::shared_ptr<AbstractCellProperty>
177         ↳ p_state(CellPropertyRegistry::Instance()→Get<WildTypeCellMutationState>());
178     boost::shared_ptr<AbstractCellProperty>
179         ↳ p_Stem(CellPropertyRegistry::Instance()→Get<StemCellProliferativeType>());
180     boost::shared_ptr<AbstractCellProperty> p_Transit(
181
182             ↳ CellPropertyRegistry::Instance()→Get<TransitCellProliferativeType>()
183     boost::shared_ptr<AbstractCellProperty> p_PostMitotic(
184
185             ↳ CellPropertyRegistry::Instance()→Get<DifferentiatedCellProliferativeType>()
186
187 //iterate through supplied seed range, executing one simulation per seed
188     for (unsigned seed = startSeed; seed ≤ endSeed; seed++)
189     {
190         //initialise SimulationTime (permits cellcyclemodel setup)
191         SimulationTime::Instance()→SetStartTime(0.0);
192
193         //Reseed the RNG with the required seed
194         p_RNG→Reseed(seed);
195
196         //unsigned numberStem =
197             ↳ int(std::round(p_RNG→NormalRandomDeviate(stemMean,
198                 ↳ stemStd)));
199         unsigned numberProgenitors =
200             ↳ int(std::round(p_RNG→NormalRandomDeviate(progenitorMean,
201                 ↳ progenitorStd)));
202         unsigned numberStem = int(std::round(numberProgenitors /
203             ↳ stemDivisor));
204
205         std::vector<CellPtr> stems;
206         std::vector<CellPtr> cells;
207
208         for (unsigned i = 0; i < numberStem; i++)
209         {
210             WanStemCellCycleModel* p_stem_model = new
211                 ↳ WanStemCellCycleModel;
212             p_stem_model→SetDimension(2);

```

```

202     p_stem_model→SetModelParameters(stemGammaShift,
203         ↳ stemGammaShape, stemGammaScale, stemOffspringParams);
204
205     CellPtr p_cell(new Cell(p_state, p_stem_model));
206     p_cell→InitialiseCellCycleModel();
207     stems.push_back(p_cell);
208     cells.push_back(p_cell);
209 }
210
211 for (unsigned i = 0; i < numberProgenitors; i++)
212 {
213     double currTil = p_RNG→ranf() * cmzResidencyTime;
214
215     HeCellCycleModel* p_prog_model = new HeCellCycleModel;
216     p_prog_model→SetDimension(2);
217     p_prog_model→SetModelParameters(currTil, mitoticModePhase2,
218         ↳ mitoticModePhase2 + mitoticModePhase3, pPP1,
219             pPD1, pPP2, pPD2, pPP3,
220                 ↳ pPD3);
221     p_prog_model→EnableKillSpecified();
222
223     CellPtr p_cell(new Cell(p_state, p_prog_model));
224     p_cell→InitialiseCellCycleModel();
225     cells.push_back(p_cell);
226 }
227
228 //Generate 1x#cells mesh for abstract colony
229 HoneycombMeshGenerator generator(1, (numberProgenitors +
230     ↳ numberStem));
231 MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
232 NodesOnlyMesh<2> mesh;
233 mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);
234
235 //Setup cell population
236 boost::shared_ptr<NodeBasedCellPopulation<2>> cell_population(new
237     ↳ NodeBasedCellPopulation<2>(mesh, cells));
238
239     ↳ cell_population→AddCellPopulationCountWriter<CellProliferativeTypesCount>();
240
241 //Give Wan stem cells the population & base stem pop size
242 for (auto p_cell : stems)
243 {
244

```

```

238     WanStemCellCycleModel* p_cycle_model =
239         → dynamic_cast<WanStemCellCycleModel*>(p_cell→GetCellCycleModel());
240     p_cycle_model→EnableExpandingStemPopulation(numberStem,
241         → cell_population);
242 }
243
244 //Setup simulator & run simulation
245 boost::shared_ptr<OffLatticeSimulationPropertyStop<2>>
246     → p_simulator(
247         new
248             → OffLatticeSimulationPropertyStop<2>(*cell_population));
249     p_simulator→SetStopProperty(p_Transit); //simulation to stop if
250         → no RPCs are left
251     p_simulator→SetDt(1);
252     p_simulator→SetOutputDirectory(directoryString + "/Seed" +
253         → std::to_string(seed) + "Results");
254     p_simulator→SetEndTime(8568); // 360dpf - 3dpf simulation start
255         → time
256     p_simulator→Solve();
257
258 //Reset for next simulation
259 SimulationTime::Destroy();
260 cell_population.reset();
261 }
262
263 p_RNG→Destroy();
264
265 return exit_code;
266 }
267 ;

```

15.1.6 /python_fixtures/He_output_fixture.py

```

1 import multiprocessing
2 import os
3 import subprocess
4 import datetime
5
6 import numpy as np
7 from imageio.plugins._bsdf import BsdfSerializer
8
9 executable = '/home/main/chaste_build/projects/ISP/apps/HeSimulator'

```

```

10
11 if not(os.path.isfile(executable)):
12     raise Exception('Could not find executable: ' + executable)
13
14 #####
15 # SIMULATION PARAMETERS
16 #####
17
18 #Define start and end RNG seeds; determines:
19 #No. lineages per loss function run
20 #unique sequence of RNG results for each lineage
21 start_seed = 0
22 end_seed_counts = 9999
23 end_seed_events = 999
24 run_modes = [0,1,2] #1=deterministic mode, 0=refit stochastic mode,
    ↪ 2=original fit
25 debug_output = 0 #0=off;1=on
26
27 count_directory = "HeCounts"
28 count_filenames = ["induction", "wan", "ath5", "validate"]
29 event_directory = "HeModeEvent"
30 event_filenames = ["inductionMode", "validateMode"]
31 induction_times = [24, 32, 48]
32
33 #####
34 #GLOBAL MODEL PARAMETERS
35 #####
36
37 #Values defining different marker induction timepoints & relative start
    ↪ time of TiL counter
38 earliest_lineage_start_time = 23.0 #RPCs begin to enter "He model regime"
    ↪ nasally at 23hpf
39 latest_lineage_start_time = 39.0 #last temporal retinal RPC has entered
    ↪ "He model regime" at 39hpf
40 counts_end_time = 72.0
41 events_end_time = 80.0
42 wan_residency_time = 17.0
43
44 #####
45 # SPECIFIC MODEL PARAMETERS - THETAHAT
46 #####
47
48 #These parameters are the results of the SPSA optimisation fixture

```

```

49
50 #STOCHASTIC MITOTIC MODE
51 #HE ORIGINAL PARAMETERS
52 #mitotic mode per-phase probabilities
53 #3-phase mitotic mode time periodisation
54 o_mitotic_mode_phase_2 = 8 #These are phase lengths, so
55 o_mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
56 o_phase_1_pPP = 1.0
57 o_phase_1_pPD = 0.0
58 o_phase_2_pPP = 0.2
59 o_phase_2_pPD = 0.4
60 o_phase_3_pPP = 0.2
61 o_phase_3_pPD = 0.0
62
63 #SPSA REFIT PARAMETERS
64 #mitotic mode per-phase probabilities
65 #3-phase mitotic mode time periodisation
66 mitotic_mode_phase_2 = 4.1483 #These are phase lengths, so
67 mitotic_mode_phase_3 = 11.6416 #Phase3 boundary = mmp2 + mmp3
68 phase_1_pPP = 1.0
69 phase_1_pPD = 0.0
70 phase_2_pPP = 0.1959
71 phase_2_pPD = 0.5168
72 phase_3_pPP = 0.2934
73 phase_3_pPD = 0.0
74 he_model_params = 15
75
76 #DETERMINISTIC MITOTIC MODE
77 #Phase boundary shift parameters
78 phase_1_shape = 3.7371
79 phase_1_scale = 1.8114
80 phase_2_shape = 2.5769
81 phase_2_scale = 1.6814
82 phase_sister_shift_widths = 1.6326
83 phase_offset = 1.2333
84 det_model_params = 12
85
86 original_theta_string = str(o_mitotic_mode_phase_2) + " " +
    ↵ str(o_mitotic_mode_phase_3) + " " + str(o_phase_1_pPP) + " " +
    ↵ str(o_phase_1_pPD) + " " + str(o_phase_2_pPP) + " " +
    ↵ str(o_phase_2_pPD) + " " + str(o_phase_3_pPP) + " " +
    ↵ str(o_phase_3_pPD)

```

```

87 stochastic_theta_string = str(mitotic_mode_phase_2) + " " +
→ str(mitotic_mode_phase_3) + " " +str(phase_1_pPP) + " " +
→ str(phase_1_pPD) + " " + str(phase_2_pPP) + " " + str(phase_2_pPD) +
→ " " + str(phase_3_pPP) + " " + str(phase_3_pPD)
88 deterministic_theta_string = str(phase_1_shape) + " " +
→ str(phase_1_scale) + " " + str(phase_2_shape) + " " +
→ str(phase_2_scale) + " " + str(phase_sister_shift_widths) + " " +
→ str(phase_offset)

89
90 def main():
91
92     command_list = []
93
94     for m in range(0,len(run_modes)):
95         curr_list = []
96
97         run_mode_string = str(run_modes[m])
98         deterministic_mode = run_modes[m]
99         if run_modes[m] == 2:
100             deterministic_mode = 0
101         command_count = 0 #for iterating seed numbers
102         base_command = executable
103
104         #induction count commands
105         for i in range(0,len(induction_times)):
106
107             output_mode = 0 #0=lineage counts;1=mitotic event
→ logging;2=sequence sampling
108             fixture = 0 #0=He 2012;1=Wan 2016
109             curr_start_seed = start_seed + command_count *
→ (end_seed_counts+1)
110             curr_end_seed = curr_start_seed + end_seed_counts
111             ath5founder = 0
112
113             command = base_command+" "\n
114                 +count_directory+" "\n
115
→ +count_filenames[0]+str(induction_times[i])+SDMode+run_
→ "\n
116             +str(output_mode)+" "\n
117             +str(deterministic_mode)+" "\n
118             +str(fixture)+" "\n
119             +str(ath5founder)+" "

```

```

120             +str(debug_output)+" "\n
121             +str(curr_start_seed)+" "\n
122             +str(curr_end_seed)+" "\n
123             +str(induction_times[i])+" "\n
124             +str(earliest_lineage_start_time)+" "\n
125             +str(latest_lineage_start_time)+" "\n
126             +str(counts_end_time)+" "
127         curr_list.append(command)
128         command_count += 1
129
130     #wan command
131     output_mode = 0
132     fixture = 1
133     curr_start_seed = start_seed + command_count *
134         ↪ (end_seed_counts+1)
135     curr_end_seed = curr_start_seed + end_seed_counts
136     ath5founder = 0
137     wan_command = base_command+" \
138             +count_directory+" "\n
139             +count_filenames[1]+"SDMode"+run_mode_string+" "\n
140             +str(output_mode)+" "\n
141             +str(deterministic_mode)+" "\n
142             +str(fixture)+" "\n
143             +str(ath5founder)+" "\n
144             +str(debug_output)+" "\n
145             +str(curr_start_seed)+" "\n
146             +str(curr_end_seed)+" "\n
147             +str(0)+" "\n
148             +str(0)+" "\n
149             +str(wan_residency_time)+" "\n
150             +str(counts_end_time)+" "
151         curr_list.append(wan_command)
152         command_count += 1
153
154     #wan- 17 hr constraint- no "shadow RPCs"
155     curr_start_seed = start_seed + command_count *
156         ↪ (end_seed_counts+1)
157     curr_end_seed = curr_start_seed + end_seed_counts
158     wan_command = base_command+" \
159             +count_directory+" "\n
160             ↪ +count_filenames[1]+"NoShadSDMode"+run_mode_string+" "
161             "\n"

```

```

159          +str(output_mode)+" "\n
160          +str(deterministic_mode)+" "\n
161          +str(fixture)+" "\n
162          +str(ath5founder)+" "\n
163          +str(debug_output)+" "\n
164          +str(curr_start_seed)+" "\n
165          +str(curr_end_seed)+" "\n
166          +str(0)+" "\n
167          +str(0)+" "\n
168          +str(wan_residency_time)+" "\n
169          +str(wan_residency_time)+" "
170      curr_list.append(wan_command)
171      command_count += 1
172
173 #ath5 command
174 output_mode = 0
175 fixture = 2
176 curr_start_seed = start_seed + command_count *
177     ↪ (end_seed_counts+1)
178 curr_end_seed = curr_start_seed + end_seed_counts
179 ath5founder = 1
180
181 ath5_command = base_command+" "\n
182     +count_directory+" "\n
183     +count_filenames[2]+"SDMode"+run_mode_string+" "\n
184     +str(output_mode)+" "\n
185     +str(deterministic_mode)+" "\n
186     +str(fixture)+" "\n
187     +str(ath5founder)+" "\n
188     +str(debug_output)+" "\n
189     +str(curr_start_seed)+" "\n
190     +str(curr_end_seed)+" "\n
191     +str(0)+" "\n
192     +str(earliest_lineage_start_time)+" "\n
193     +str(latest_lineage_start_time)+" "\n
194     +str(counts_end_time)+" "
195 curr_list.append(ath5_command)
196 command_count += 1
197
198 #validate counts command
199 curr_start_seed = start_seed + command_count *
200     ↪ (end_seed_counts+1)
201 curr_end_seed = curr_start_seed + end_seed_counts

```

```

200     ath5founder = 0
201
202     validate_command = base_command+ " " \
203         +count_directory+ " " \
204         +count_filenames[3]+ "SDMode"+run_mode_string+ " " \
205         +str(output_mode)+ " " \
206         +str(deterministic_mode)+ " " \
207         +str(fixture)+ " " \
208         +str(ath5founder)+ " " \
209         +str(debug_output)+ " " \
210         +str(curr_start_seed)+ " " \
211         +str(curr_end_seed)+ " " \
212         +str(0)+ " " \
213         +str(earliest_lineage_start_time)+ " " \
214         +str(latest_lineage_start_time)+ " " \
215         +str(counts_end_time)+ " "
216     curr_list.append(validate_command)
217     command_count += 1
218
219     #mitotic mode rate command
220     output_mode = 1
221     fixture = 0
222     curr_start_seed = start_seed + command_count *
223         → (end_seed_counts+1)
224     curr_end_seed = curr_start_seed + end_seed_events
225     mode_rate_command = base_command+ " " \
226         +event_directory+ " " \
227         +event_filenames[0]+ "SDMode"+run_mode_string+ " " \
228         +str(output_mode)+ " " \
229         +str(deterministic_mode)+ " " \
230         +str(fixture)+ " " \
231         +str(ath5founder)+ " " \
232         +str(debug_output)+ " " \
233         +str(curr_start_seed)+ " " \
234         +str(curr_end_seed)+ " " \
235         +str(earliest_lineage_start_time)+ " " \
236         +str(earliest_lineage_start_time)+ " " \
237         +str(latest_lineage_start_time)+ " " \
238         +str(events_end_time)+ " "
239     curr_list.append(mode_rate_command)
240     command_count += 1
241

```

```

242     #validate rate command
243     fixture = 2
244     curr_start_seed = start_seed + command_count *
245         ↵ (end_seed_counts+1)
246     curr_end_seed = curr_start_seed + end_seed_events
247     validate_rate_command = base_command+" \"\
248         +event_directory+" \"\
249         +event_filenames[1]+"SDMode"+run_mode_string+" \"\
250         +str(output_mode)+" \"\
251         +str(deterministic_mode)+" \"\
252         +str(fixture)+" \"\
253         +str(ath5founder)+" \"\
254         +str(debug_output)+" \"\
255         +str(curr_start_seed)+" \"\
256         +str(curr_end_seed)+" \"\
257         +str(0)+" \"\
258         +str(earliest_lineage_start_time)+" \"\
259         +str(latest_lineage_start_time)+" \"\
260         +str(events_end_time)+" "
261
262     curr_list.append(validate_rate_command)
263     command_count += 1
264
265     for i in range(0,len(curr_list)):
266         if run_modes[m] == 2:
267             curr_list[i] = curr_list[i] + original_theta_string
268         if run_modes[m] == 0:
269             curr_list[i] = curr_list[i] + stochastic_theta_string
270         if run_modes[m] == 1:
271             curr_list[i] = curr_list[i] + deterministic_theta_string
272
273     command_list = command_list + curr_list
274
275     # Use processes equal to the number of cpus available
276     cpu_count = multiprocessing.cpu_count()
277
278     print(command_list)
279
280     print("Starting simulations with " + str(cpu_count) + " processes")
281
282     # Generate a pool of workers
283     pool = multiprocessing.Pool(processes=cpu_count)

```

```

284     # Pass the list of bash commands to the pool, block until pool is
285     # complete
286     pool.map(execute_command, command_list, 1)
287
288 # This is a helper function for run_simulation that runs bash commands in
289 # separate processes
290 def execute_command(cmd):
291     print("Executing command: " + cmd)
292     return subprocess.call(cmd, shell=True)
293
294 if __name__ == "__main__":
295     main()

```

15.1.7 /python_fixtures/Kolmogorov_fixture.py

```

1 import multiprocessing
2 import os
3 import subprocess
4 import datetime
5 import numpy as np
6 from imageio.plugins._bsdf import BsdfSerializer
7
8 gomes_executable =
9     '/home/main/chaste_build/projects/ISP/apps/GomesSimulator'
10 he_executable = '/home/main/chaste_build/projects/ISP/apps/HeSimulator'
11 boije_executable =
12     '/home/main/chaste_build/projects/ISP/apps/BoijeSimulator'
13
14 empirical_data =
15     '/home/main/git/chaste/projects/ISP/empirical_data/empirical_lineages.csv'
16
17 if not(os.path.isfile(gomes_executable)):
18     raise Exception('Could not find executable: ' + gomes_executable)
19 if not(os.path.isfile(he_executable)):
20     raise Exception('Could not find executable: ' + he_executable)
21 if not(os.path.isfile(boije_executable)):
22     raise Exception('Could not find executable: ' + boije_executable)
23
24 #####
25 # SIMULATION PARAMETERS

```

```

23 #####
24
25 #Define start and end RNG seeds; determines:
26 #No. lineages per loss function run
27 #unique sequence of RNG results for each lineage
28
29 directory = "KolmogorovSequences"
30 empirical_sequences_name = "E0sequences"
31 output_mode = 2 #sequence sampler
32 debug_output = 0 #0=off;1=on
33 start_seed = 0
34 end_seed = 9999
35
36 #####
37 #GLOBAL MODEL PARAMETERS
38 #####
39
40 number_traversal_lineages = 60
41
42 #####
43 # SPECIFIC MODEL PARAMETERS
44 #####
45
46 #GOMES MODEL
47 g_end_time = 480
48 g_normal_mean = 3.9716
49 g_normal_sigma = .32839
50 g_pPP = .055
51 g_pPD = .221
52 g_pBC = .128
53 g_pAC = .106
54 g_pMG = .028
55
56 g_string = gomes_executable + " " + directory + " Gomes " +
   ↵ str(output_mode) + " " + str(debug_output) + " " + str(start_seed) +
   ↵ " " + str(end_seed) + " " + str(g_end_time) + " " +
   ↵ str(g_normal_mean) + " " + str(g_normal_sigma) + " " + str(g_pPP) + " "
   ↵ " " + str(g_pPD) + " " + str(g_pBC) + " " + str(g_pAC) + " " +
   ↵ str(g_pMG)
57
58 #HE MODEL - GENERAL
59 h_fixture = 2
60 h_ath5founder = 0

```

```

61 h_start_time = 0
62 h_end_time = 80
63
64 #HE MODEL - STOCHASTIC
65 h_deterministic_mode = 0
66 h_mitotic_mode_phase_2 = 8 #These are phase lengths, so
67 h_mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
68 h_phase_1_pPP = 1.0
69 h_phase_1_pPD = 0.0
70 h_phase_2_pPP = 0.2
71 h_phase_2_pPD = 0.4
72 h_phase_3_pPP = 0.2
73 h_phase_3_pPD = 0.0
74
75 h_string = he_executable + " " + directory + " He " + str(output_mode) +
    " " + str(h_deterministic_mode) + " " + str(h_fixture) + " " +
    str(h_ath5founder) + " " + str(debug_output) + " " + str(start_seed)
    + " " + str(end_seed) + " " + str(h_start_time) + " " +
    str(h_start_time) + " " + str(h_start_time+1) + " " + str(h_end_time)
    + " " + str(h_mitotic_mode_phase_2) + " " +
    str(h_mitotic_mode_phase_3) + " " +str(h_phase_1_pPP) + " " +
    str(h_phase_1_pPD) + " " + str(h_phase_2_pPP) + " " +
    str(h_phase_2_pPD) + " " + str(h_phase_3_pPP) + " " +
    str(h_phase_3_pPD)
76
77 #HE MODEL -STOCHASTIC (REFIT)
78 hr_mitotic_mode_phase_2 = 4.1483 #These are phase lengths, so
79 hr_mitotic_mode_phase_3 = 11.6416 #Phase3 boundary = mmp2 + mmp3
80 hr_phase_1_pPP = 1.0
81 hr_phase_1_pPD = 0.0
82 hr_phase_2_pPP = 0.1959
83 hr_phase_2_pPD = 0.5168
84 hr_phase_3_pPP = 0.2934
85 hr_phase_3_pPD = 0.0
86

```

```

87 hr_string = he_executable + " " + directory + " HeRefit " +
→ str(output_mode) + " " + str(h_deterministic_mode) + " " +
→ str(h_fixture) + " " + str(h_ath5founder) + " " + str(debug_output) +
→ " " + str(start_seed) + " " + str(end_seed) + " " + str(h_start_time)
→ + " " + str(h_start_time) + " " + str(h_start_time+1) + " " +
→ str(h_end_time) + " " + str(hr_mitotic_mode_phase_2) + " " +
→ str(hr_mitotic_mode_phase_3) + " " + str(hr_phase_1_pPP) + " " +
→ str(hr_phase_1_pPD) + " " + str(hr_phase_2_pPP) + " " +
→ str(hr_phase_2_pPD) + " " + str(hr_phase_3_pPP) + " " +
→ str(hr_phase_3_pPD)

88
89 #HE MODEL - DETERMINISTIC ALTERNATIVE
90 d_deterministic_mode = 1
91 #Phase boundary shift parameters
92 d_phase_1_shape = 3.7371
93 d_phase_1_scale = 1.8114
94 d_phase_2_shape = 2.5769
95 d_phase_2_scale = 1.6814
96 d_phase_sister_shift_widths = 1.6326
97 d_phase_offset = 1.2333

98
99 d_string = he_executable + " " + directory + " Deterministic " +
→ str(output_mode) + " " + str(d_deterministic_mode) + " " +
→ str(h_fixture) + " " + str(h_ath5founder) + " " + str(debug_output) +
→ " " + str(start_seed) + " " + str(end_seed) + " " + str(h_start_time)
→ + " " + str(h_start_time) + " " + str(h_start_time+1) + " " +
→ str(h_end_time) + " " + str(d_phase_1_shape) + " " +
→ str(d_phase_1_scale) + " " + str(d_phase_2_shape) + " " +
→ str(d_phase_2_scale) + " " + str(d_phase_sister_shift_widths) + " " +
→ str(d_phase_offset)

100
101 #BOIJE MODEL
102 b_end_generation = 250
103 b_phase_2_generation = 3
104 b_phase_3_generation = 5
105 b_pAtoh7 = .32
106 b_pPtf1a = .3
107 b_png = .8
108

```

```

109 b_string = boije_executable + " " + directory + " Boije " +
→   str(output_mode) + " " + str(debug_output) + " " + str(start_seed) +
→   " " + str(end_seed) + " " + str(b_end_generation) + " " +
→   str(b_phase_2_generation) + " " + str(b_phase_3_generation) + " " +
→   str(b_pAtoh7) + " " + str(b_pPtf1a) + " " + str(b_png)

110
111 #setup the log file, appending to any existing results
112 e_filename =
→   "/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/"
→   +directory +"/" + empirical_sequences_name
113 os.makedirs(os.path.dirname(e_filename), exist_ok=True)

114
115 e_file = open(e_filename, "w")
116 e_file.write("Entry\tSequence\n")

117
118 def main():

119     command_list = []
120     command_list.append(g_string)
121     command_list.append(h_string)
122     command_list.append(hr_string)
123     command_list.append(d_string)
124     command_list.append(b_string)

125
126     # Use processes equal to the number of cpus available
127     cpu_count = multiprocessing.cpu_count()

128
129     print(command_list)

130
131     print("Starting simulations with " + str(cpu_count) + " processes")

132
133     # Generate a pool of workers
134     pool = multiprocessing.Pool(processes=cpu_count)

135
136     # Pass the list of bash commands to the pool, block until pool is
→       complete
137     pool.map(execute_command, command_list, 1)

138
139     traverse_lineages(empirical_data)

140
141
142 def traverse_lineages(data_filename):

143
144     mode_sequences = [None]*(end_seed+1)

```

```

145
146     #load lineage tracing data
147     e_data = np.loadtxt(data_filename, skiprows = 1, usecols = (0,1,2,3))
148
149     #reproducible RNG
150     p_RNG = np.random.RandomState(seed = 0)
151
152     k=0
153
154     for curr_seed in range (start_seed,end_seed+1):
155         random_lineage_number =
156             ↪ p_RNG.randint(1,number_traversal_lineages)
157         lineage_events =
158             ↪ np.array(e_data[np.where(e_data[:,0]==random_lineage_number)])
159
160         #find mitotic mode of first event and write to log
161         current_event = 1
162         mitotic_mode =
163             ↪ int(lineage_events[np.where(lineage_events[:,1]==current_event)][:,3])
164         mode_sequences[k] = f'{mitotic_mode:.0f}'
165
166         while mitotic_mode != 2:
167             if mitotic_mode == 1 and p_RNG.random_sample() < .5: break
168
169             child_events =
170                 ↪ lineage_events[np.where(lineage_events[:,2]==current_event)]
171             random_child_row =
172                 ↪ p_RNG.randint(0,np.ma.size(child_events,0))
173             current_event= child_events[random_child_row,1]
174             mitotic_mode = child_events[random_child_row,3]
175             mode_sequences[k] = mode_sequences[k] + f'{mitotic_mode:.0f}'

176             e_file.write(str(k+1)+"\t"+mode_sequences[k]+\n)
177             k += 1
178
179     return mode_sequences
180
181
182     # This is a helper function for run_simulation that runs bash commands in
183     ↪ separate processes
184     def execute_command(cmd):
185         print("Executing command: " + cmd)
186         return subprocess.call(cmd, shell=True)

```

```
182 if __name__ == "__main__":
183     main()
```

15.1.8 /python_fixtures/SPSA_fixture.py

```
1 import multiprocessing
2 import os
3 import subprocess
4 import datetime
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8 from scipy.stats import bernoulli
9 from fileinput import filename
10 from statsmodels.sandbox.distributions.gof_new import a_st70_upp
11
12 executable = '/home/main/chaste_build/projects/ISP/apps/HeSimulator'
13
14 if not(os.path.isfile(executable)):
15     raise Exception('Could not find executable: ' + executable)
16
17 #####
18 # SPSA COEFFICIENTS
19 #####
20
21 a_s = .025
22 c_s = .1
23
24 a_d = .0019
25 c_d = .1
26
27 A = 20 #10% of expected # iterations
28 alpha = .602
29 gamma = .101
30
31 #####
32 # SPSA & AIC UTILITY VARS
33 #####
34
35 max_iterations = 199
```

```

36 number_comparisons = 3030 #total number of comparison points between
   ↵ model output and empirical data (1000 per induction time, 10 per rate
   ↵ mode)
37 number_comparisons_per_induction = 1000
38 error_samples = 5000 #number of samples to draw when estimating
   ↵ plausibility interval for simulations
39
40 #####
41 # SIMULATION PARAMETERS
42 #####
43
44 #Define start and end RNG seeds; determines:
45 #No. lineages per loss function run
46 #unique sequence of RNG results for each lineage
47 start_seed = 0
48 end_seed = 249
49 rate_end_seed = 99
50 directory_name = "SPSA"
51 file_name_he = "HeSPSA"
52 file_name_det = "DetSPSA"
53 log_name = "HeSPSAOutput"
54 deterministic_modes = [0, 1] #1=det. mode enabled
55 count_output_mode = 0 #0=lineage counts;1=mitotic event
   ↵ logging;2=sequence sampling
56 event_output_mode = 1
57 fixture = 0 #0=He 2012;1=Wan 2016
58 debug_output = 0 #0=off;1=on
59 ath5founder = 0 #0=no morpholino 1=ath5 morpholino
60
61 #####
62 #GLOBAL MODEL PARAMETERS
63 #####
64
65 #Values defining different marker induction timepoints & relative start
   ↵ time of TiL counter
66 earliest_lineage_start_time = 23.0 #RPCs begin to enter "He model regime"
   ↵ nasally at 23hpf
67 latest_lineage_start_time = 39.0 #last temporal retinal RPC has entered
   ↵ "He model regime" at 39hpf
68 induction_times = [ 24, 32, 48 ]
69 end_time = 72.0
70 rate_end_time = 80
71

```

```

72 #####
73 # SPECIFIC MODEL PARAMETERS - THETAHAT
74 #####
75
76 #STOCHASTIC MITOTIC MODE
77 #mitotic mode per-phase probabilities
78 #3-phase mitotic mode time periodisation
79 mitotic_mode_phase_2 = 8 #These are phase lengths, so
80 mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
81 phase_1_pPP = 1.0
82 phase_1_pPD = 0.0
83 phase_2_pPP = 0.2
84 phase_2_pPD = 0.4
85 phase_3_pPP = 0.2
86 phase_3_pPD = 0.0
87 he_model_params = 15
88
89 #DETERMINISTIC MITOTIC MODE
90 #Phase boundary shift parameters
91 phase_1_shape = 3
92 phase_1_scale = 2
93 phase_2_shape = 2
94 phase_2_scale = 2
95 phase_sister_shift_widths = .25
96 phase_offset = 0
97 det_model_params = 13
98
99 #array "theta_spsa" is manipulated during SPSA calculations, these are
  ↳ starting point references
100 stochastic_theta_zero = np.array([mitotic_mode_phase_2,
  ↳ mitotic_mode_phase_3, phase_2_pPP, phase_2_pPD, phase_3_pPP ])
101 determinisitic_theta_zero = np.array([phase_1_shape, phase_1_scale,
  ↳ phase_2_shape, phase_2_scale, phase_sister_shift_widths,
  ↳ phase_offset])
102
103 #scales ak gain sequence for probability variables
104 prob_scale_vector = np.array([ 1, 1, .1, .1, .1])
105 shift_scale_vector = np.array([1, 1, 1, 1, 1, 3])
106
107 #####
108 # HE ET AL EMPIRICAL RESULTS
109 #####
110

```

```

111 count_bin_sequence = np.arange(1,32,1)
112 count_x_sequence = np.arange(1,31,1)
113 count_trim_value = 30
114
115 rate_bin_sequence = np.arange(30,85,5)
116 rate_x_sequence = np.arange(30,80,5)
117 rate_trim_value = 10
118
119 #Count probability arrays
120 raw_counts =
    ↵ np.loadtxt('/home/main/git/chaste/projects/ISP/empirical_data/empirical_counts.csv',
    ↵ skiprows=1, usecols=(3,4,5,6,7,8,9,10)) #collect the per-cell-type
    ↵ counts
121 raw_counts_24 = raw_counts[0:64,:]
122 raw_counts_32 = raw_counts[64:233,:]
123 raw_counts_48 = raw_counts[233:396,:]
124
125 prob_24,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_24,axis=1),count_bin_sequence,density=True)
    ↵ #obtain probability density histogram for counts, retabulating by
    ↵ summing across types
126 prob_32,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_32,axis=1),count_bin_sequence,density=True)
127 prob_48,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_48,axis=1),count_bin_sequence,density=True)
128
129 #extend histogram to 1000 - allows large lineages of some iterates to be
    ↵ included in AIC comparison
130 prob_empirical_24 = np.concatenate([prob_24,
    ↵ np.zeros(int(number_comparisons_per_induction) - prob_24.size)])
131 prob_empirical_32 = np.concatenate([prob_32,
    ↵ np.zeros(int(number_comparisons_per_induction) - prob_32.size)])
132 prob_empirical_48 = np.concatenate([prob_48,
    ↵ np.zeros(int(number_comparisons_per_induction) - prob_48.size)])
133
134 count_prob_list = [prob_empirical_24, prob_empirical_32,
    ↵ prob_empirical_48]
135
136 #no. of lineages observed per induction timepoint/event group
137 lineages_sampled_24 = 64
138 lineages_sampled_32 = 169
139 lineages_sampled_48 = 163
140 lineages_sampled_events = 60

```

```

141 lineages_sampled_list =
142     ↳ [lineages_sampled_24,lineages_sampled_32,lineages_sampled_48]
143
144 #Mitotic mode rate probability arrays
145 raw_events =
146     ↳ np.loadtxt('/home/main/git/chaste/projects/ISP/empirical_data/empirical_lineages.
147     ↳ skiprows=1, usecols=(3,5,8))
148 observed_events = raw_events[np.where(raw_events[:,2]==1)] #exclude any
149     ↳ mitosis whose time was too early for recording
150
151 observed_PP = observed_events[np.where(observed_events[:,0]==0)]
152 observed_PD = observed_events[np.where(observed_events[:,0]==1)]
153 observed_DD = observed_events[np.where(observed_events[:,0]==2)]
154
155 histo_PP,bin_edges =
156     ↳ np.histogram(observed_PP,rate_bin_sequence,density=False)
157 histo_PD,bin_edges =
158     ↳ np.histogram(observed_PD,rate_bin_sequence,density=False)
159 histo_DD,bin_edges =
160     ↳ np.histogram(observed_DD,rate_bin_sequence,density=False)
161
162 #hourly per-lineage probabilities- NOT probability density function
163 prob_empirical_PP = np.array((histo_PP/lineages_sampled_events)/5)
164 prob_empirical_PD = np.array((histo_PD/lineages_sampled_events)/5)
165 prob_empirical_DD = np.array((histo_DD/lineages_sampled_events)/5)
166
167 event_prob_list = [prob_empirical_PP,prob_empirical_PD,prob_empirical_DD]
168
169 #setup the log file, appending to any existing results
170 log_filename =
171     ↳ "/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/"
172     ↳ +directory_name +"/" + log_name
173 os.makedirs(os.path.dirname(log_filename), exist_ok=True)
174
175 log = open(log_filename,"w")
176
177 #plotting utility stuff
178 #interactive plot mode
179 plt.ion()
180
181 #setup new iterate plot
182 fig, ((plt0, plt1, plt2),(plt3, plt4, plt5)) =
183     ↳ plt.subplots(2,3,figsize=(12,6))
184 plot_list = [plt0,plt1,plt2,plt3,plt4,plt5]

```

```

174
175
176 def main():
177     global theta_spsa,
178         ↳ a,c,file_name,scale_vector,end_seed,rate_end_seed,alpha,gamma
179
180     for m in range(0,len(deterministic_modes)):
181
182         #traceable RNG
183         p_RNG = np.random.RandomState(seed=786)
184
185         deterministic_mode = deterministic_modes[m]
186         if deterministic_mode == 0:
187             theta_spsa = stochastic_theta_zero
188             a = a_s
189             c = c_s
190             file_name = file_name_he
191             scale_vector = prob_scale_vector
192             now = datetime.datetime.now()
193             log.write("Began SPSA optimisation of He model @\n" +
194             ↳ str(datetime.datetime.now()) + "\n")
195             log.write("k\tphase2\tphase3\tPP2\tPD2\tPP3\n")
196             number_params = he_model_params
197             if deterministic_mode == 1:
198                 theta_spsa = determinisitic_theta_zero
199                 a = a_d
200                 c = c_d
201                 file_name = file_name_det
202                 scale_vector = shift_scale_vector
203                 now = datetime.datetime.now()
204                 log.write("Began SPSA optimisation of deterministic model
205                 ↳ @\n" + str(datetime.datetime.now()) + "\n")
206                 log.write("k\tP1Sh\tP1Sc\tP2Sh\tP2Sc\tSisterShift\tOffset\n")
207                 number_params = det_model_params
208
209                 #SPSA algorithm iterator k starts at 0
210                 k=0
211
212                 while k <= max_iterations:
213                     if k == 0:
214                         end_seed = 249
215                         rate_end_seed = 99

```

```

214     #@defined iterate, increase # of seed to decrease RNG noise
215     #> to low level
216     if k == 169:
217         end_seed = 999
218         rate_end_seed = 249
219
220     #@defined iterate, increase # of seeds to decrease RNG noise
221     #> to close to nil, switch to asymptotically optimal alpha
222     #> and gamma
223     if k == 189:
224         end_seed = 4999
225         rate_end_seed = 1249
226         alpha = 1
227         gamma = (1/6)
228
229     #write the parameter set to be evaluated to file
230     if deterministic_mode == 0: log.write(str(k) + "\t" +
231     #> str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" +
232     #> str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" +
233     #> str(theta_spsa[4]) + "\n")
234     if deterministic_mode == 1: log.write(str(k) + "\t" +
235     #> str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" +
236     #> str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" +
237     #> str(theta_spsa[4]) + "\t" + str(theta_spsa[5]) + "\n")
238
239     #populate the deltak perturbation vector with samples from a
240     #> .5p bernoulli +1 -1 distribution
241     delta_k = bernoulli.rvs(.5, size=theta_spsa.size,
242     #> random_state=p_RNG)
243     delta_k[delta_k == 0] = -1
244
245     ak = a / ((A + k + 1)**alpha) #calculate ak from gain
246     #> sequence
247     scaled_ak = ak * scale_vector #scale ak appropriately for
248     #> parameters expressed in hrs & percent
249
250     ck = c / ((k + 1)**gamma) #calculate ck from gain sequence
251     scaled_ck = ck * scale_vector
252
253     #Project theta_spsa into space bounded by ck to allow
254     #> gradient sampling at bounds of probability space

```

```

242     projected_theta = project_theta(theta_spsa, scaled_ck,
243                                     ↵ deterministic_mode)
244
245     #Calculate theta+ and theta- vectors for gradient estimate
246     theta_plus = projected_theta + scaled_ck * delta_k
247     theta_minus = projected_theta - scaled_ck * delta_k
248
249     AIC_gradient = evaluate_AIC_gradient(k, theta_plus,
250                                         ↵ theta_minus, deterministic_mode, number_params,
251                                         ↵ file_name)
252
253     ghat = ((AIC_gradient) / (2 * ck)) * delta_k
254
255     log.write("g0: " + str(ghat[0]) + "\n")
256
257     #update new theta_spsa
258     theta_spsa = theta_spsa - scaled_ak * ghat
259
260     #constrain updated theta_spsa
261     theta_spsa = project_theta(theta_spsa,
262                                 ↵ np.zeros(theta_spsa.size), deterministic_mode)
263
264     k+=1
265
266     #write final result and close log
267     if deterministic_mode == 0: log.write(str(k) + "\t" +
268                                         ↵ str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" +
269                                         ↵ str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" +
270                                         ↵ str(theta_spsa[4]) + "\n")
271     if deterministic_mode == 1: log.write(str(k) + "\t" +
272                                         ↵ str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" +
273                                         ↵ str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" +
274                                         ↵ str(theta_spsa[4]) + "\t" + str(theta_spsa[5]) + "\n")
275
276     log.close
277
278 def project_theta(theta, boundary, deterministic_mode):
279     # Required projection is onto unit triangle modified by boundary (ck
280     ↵ value)
281
282     # Values outside the lower bounds are reset first
283     # Then, for phase 2 probabilities, we project (PPa,PDa)→(PPb,PDb) such
284     ↵ that if( (PPa+ck) + (PDa+ck) >1), PPb+ck + PDb +ck = 1.
285
286     # This takes care of the upper bound

```

```

273
274     #project all negative parameters back into bounded space
275     if deterministic_mode == False:
276         for i in range(0, theta.size):
277             if i == 0:
278                 if theta[i] < boundary[i] + 4.0: theta[i] = boundary[i] +
279                     → 4.0 #project mitotic_mode_phase_2 to a minimum of 4-
280                     → below this param has no effect (due to refractory
281                     → period after first division)
282             else:
283                 if theta[i] < boundary[i]: theta[i] = boundary[i]
284
285             #if PP3 exceeds 1-boundary, project back to 1-boundary
286             if theta[4] > 1 - boundary[4]: theta[4] = 1 - boundary[4]
287
288             if theta[2] + theta[3] > 1 - boundary[2]: #if pPP2 + pPD2 gives a
289                 → total beyond the current boundary-reduced total of 1.0
290
291             if theta[2] > (1 - 2 * boundary[2]) and theta[2] ≤ theta[3]
292                 → - (1 - 2 * boundary[2]): # these (PP,PD) points will
293                 → project into negative PD space
294                 theta[2] = 1 - 2 * boundary[2]
295                 theta[3] = boundary[2]
296
297             elif theta[3] > (1 - 2 * boundary[2]) and theta[2] ≤
298                 → theta[3] - (1 - 2 * boundary[2]): # these (PP,PD) points
299                 → will project into negative PP space
300                 theta[3] = 1 - 2 * boundary[2]
301                 theta[2] = boundary[2]
302
303             else: # the (PP,PD) points can be projected onto the line PD
304                 → = -PP + 1 and modified by the boundary;
305                 v1 = [ 1, -1 ] # vector from (0,1) to (1,0)
306                 v2 = [ theta[2], theta[3] - 1 ] #vector from (0,1) to
307                     → (PP,PD)
308                 dot_product = np.dot(v1,v2)
309                 lengthv1 = 2
310                 theta[2] = (dot_product / lengthv1) - boundary[2]
311                 theta[3] = (1 - dot_product / lengthv1) - boundary[2]
312
313             if deterministic_mode == True:
314                 for i in range(0, theta.size-1): #exclude offset param

```

```

305         if theta[i] < boundary[i]: theta[i] = boundary[i] + 0.1
306             ↵ #prevents non→0 arguments for deterministic mode
307
308     #projects sister shift value such that 95% of sister shift values
309     ↵ will be less than smallest mean phase time
310     minPhase = min(theta[0]*theta[1],theta[2]*theta[3])
311     if theta[4] > minPhase/2 - boundary[4]: theta[4] = minPhase/2 -
312         ↵ boundary[4]
313
314     return theta
315
316
317 def evaluate_AIC_gradient(k, theta_plus, theta_minus, deterministic_mode,
318     ↵ number_params, file_name):
319     #Form the simulator commands for current thetas and deterministic
320     ↵ modes
321     command_list = []
322     base_command = executable
323
324     rate_settings = str(event_output_mode)+" "\
325         +str(deterministic_mode)+" "\
326         +str(fixture)+" "\
327         +str(ath5founder)+" "\
328         +str(debug_output)+" "\
329         +str(start_seed)+" "\
330         +str(rate_end_seed)+" "\
331         +str(earliest_lineage_start_time)+" "\
332         +str(earliest_lineage_start_time)+" "\
333         +str(latest_lineage_start_time)+" "\
334         +str(rate_end_time)+" "
335
336     count_settings_1 = str(count_output_mode)+" "\
337         +str(deterministic_mode)+" "\
338         +str(fixture)+" "\
339         +str(ath5founder)+" "\
340         +str(debug_output)+" "\
341         +str(start_seed)+" "\
342         +str(end_seed)+" "
343
344     count_settings_2 = str(earliest_lineage_start_time)+" "\
345         +str(latest_lineage_start_time)+" "\
346         +str(end_time)+" "
347
348     if deterministic_mode == 0:

```

```

343
344     stochastic_params_plus = str(theta_plus[0])+" "+\
345         +str(theta_plus[1])+" "+\
346         +str(phase_1_pPP)+" "+\
347         +str(phase_1_pPD)+" "+\
348         +str(theta_plus[2])+" "+\
349         +str(theta_plus[3])+" "+\
350         +str(theta_plus[4])+" "+\
351         +str(phase_3_pPD)

352
353     stochastic_params_minus = str(theta_minus[0])+" "+\
354         +str(theta_minus[1])+" "+\
355         +str(phase_1_pPP)+" "+\
356         +str(phase_1_pPD)+" "+\
357         +str(theta_minus[2])+" "+\
358         +str(theta_minus[3])+" "+\
359         +str(theta_minus[4])+" "+\
360         +str(phase_3_pPD)

361
362     command_rate_plus = base_command\
363         +" "+directory_name+" "+file_name+"RatePlus "+\
364         +rate_settings\
365         +stochastic_params_plus

366
367     command_rate_minus = base_command\
368         +" "+directory_name+" "+file_name+"RateMinus "+\
369         +rate_settings\
370         +stochastic_params_minus

371
372     command_list.append(command_rate_plus)
373     command_list.append(command_rate_minus)

374
375     for i in range(0,len(induction_times)):
376         command_plus = base_command\
377             +" "+directory_name+
378             " "+file_name+str(induction_times[i])+"Plus "+\
379             +count_settings_1\
380             +str(induction_times[i])+" "+\
381             +count_settings_2\
382             +stochastic_params_plus

383         command_minus = base_command\

```

```

384             + " "+directory_name+
385             ↵ "+file_name+str(induction_times[i])+"Minus "\n
386             +count_settings_1\
387             +str(induction_times[i])+ " "\n
388             +count_settings_2\
389             +stochastic_params_minus

390         command_list.append(command_plus)
391         command_list.append(command_minus)

392
393     if deterministic_mode == 1:

394
395         deterministic_params_plus = str(theta_plus[0])+" "\n
396             +str(theta_plus[1])+" "\n
397             +str(theta_plus[2])+" "\n
398             +str(theta_plus[3])+" "\n
399             +str(theta_plus[4])+" "\n
400             +str(theta_plus[5])

401
402         deterministic_params_minus = str(theta_minus[0])+" "\n
403             +str(theta_minus[1])+" "\n
404             +str(theta_minus[2])+" "\n
405             +str(theta_minus[3])+" "\n
406             +str(theta_minus[4])+" "\n
407             +str(theta_minus[5])

408
409         command_rate_plus = base_command\
410             +" "+directory_name+" "+ file_name+"RatePlus "\n
411             +rate_settings\
412             +deterministic_params_plus

413
414         command_rate_minus = base_command\
415             +" "+directory_name+" "+ file_name+"RateMinus "\n
416             +rate_settings\
417             +deterministic_params_minus

418
419         command_list.append(command_rate_plus)
420         command_list.append(command_rate_minus)

421
422     for i in range(0,len(induction_times)):
423         command_plus = base_command\
424             +" "+directory_name+
425             ↵ "+file_name+str(induction_times[i])+"Plus "\n

```

```

425             +count_settings_1\
426             +str(induction_times[i])+ " "\
427             +count_settings_2\
428             +deterministic_params_plus
429
430         command_minus = base_command\
431             + " "+directory_name+
432             ↳ " "+file_name+str(induction_times[i])+"Minus "\
433             +count_settings_1\
434             +str(induction_times[i])+ " "\
435             +count_settings_2\
436             +deterministic_params_minus
437
438         command_list.append(command_plus)
439         command_list.append(command_minus)
440
441     # Use processes equal to the number of cpus available
442     cpu_count = multiprocessing.cpu_count()
443
444     print("Starting simulations for iterate " + str(k) + " with " +
445           ↳ str(cpu_count) + " processes, " + str(end_seed+1) + " lineages
446           ↳ simulated for counts, " + str(rate_end_seed+1) + " lineages for
447           ↳ events")
448
449     log.flush() #required to prevent pool from jamming up log for some
450             ↳ reason
451
452     # Generate a pool of workers
453     pool = multiprocessing.Pool(processes=cpu_count)
454
455     # Pass the list of bash commands to the pool, block until pool is
456             ↳ complete
457     pool.map(execute_command, command_list, 1)
458
459     #these numpy arrays hold the individual RSS vals for timepoints +
460             ↳ rates
461     rss_plus = np.zeros(len(induction_times)+3)
462     rss_minus = np.zeros(len(induction_times)+3)
463
464     #clear the plots on the interactive figure
465     for i in range(0, len(plot_list)):
466         plt.sca(plot_list[i])

```



```

486     histo_rate_plus, bin_edges = np.histogram(mode_rate_plus[:,0],
487         ↵ rate_bin_sequence, density=False)
487     histo_rate_minus, bin_edges = np.histogram(mode_rate_minus[:,0],
488         ↵ rate_bin_sequence, density=False)

488
489     #hourly per-lineage probabilities
490     prob_histo_rate_plus = np.array(histo_rate_plus / ((rate_end_seed
491         ↵ + 1)*5))
491     prob_histo_rate_minus = np.array(histo_rate_plus /
492         ↵ ((rate_end_seed + 1)*5))

492
493     residual_plus = prob_histo_rate_plus - event_prob_list[i]
494     residual_minus = prob_histo_rate_minus - event_prob_list[i]

495
496     #PD RESIUDAL WEIGHTING
497     if i == 1:
498         rss_plus[i+3] = np.sum(1.5*np.square(residual_plus))
499         rss_minus[i+3] = np.sum(1.5*np.square(residual_minus))

500
501     else:
502         rss_plus[i+3] = np.sum(np.square(residual_plus))
503         rss_minus[i+3] = np.sum(np.square(residual_minus))

504
505     plotter(plot_list[i+3], mode_rate_plus[:,0],
506         ↵ mode_rate_minus[:,0],
507         ↵ event_prob_list[i],lineages_sampled_events, 1)

508     AIC_plus = 2 * number_params + number_comparisons *
509         ↵ np.log(np.sum(rss_plus))
510     AIC_minus = 2 * number_params + number_comparisons *
511         ↵ np.log(np.sum(rss_minus))

512
513     plt.show()
514
515         ↵ plt.savefig("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/"
516         ↵ +directory_name + "/" + "SDmode" + str(deterministic_mode) +
517         ↵ "iterate" + str(k) + ".png")

518     log.write("theta_plus:\n")
519     if deterministic_mode == 0: log.write(str(k) + "\t" +
520         ↵ str(theta_plus[0]) + "\t" + str(theta_plus[1]) + "\t" +
521         ↵ str(theta_plus[2]) + "\t" + str(theta_plus[3]) + "\t" +
522         ↵ str(theta_plus[4]) + "\n")

```

```

515     if deterministic_mode == 1: log.write(str(k) + "\t" +
516         → str(theta_plus[0]) + "\t" + str(theta_plus[1]) + "\t" +
517         → str(theta_plus[2]) + "\t" + str(theta_plus[3]) + "\t" +
518         → str(theta_plus[4]) + "\t" + str(theta_plus[5]) +"\n")
519     log.write("theta_minus:\n")
520     if deterministic_mode == 0: log.write(str(k) + "\t" +
521         → str(theta_minus[0]) + "\t" + str(theta_minus[1]) + "\t" +
522         → str(theta_minus[2]) + "\t" + str(theta_minus[3]) + "\t" +
523         → str(theta_minus[4]) + "\n")
524     if deterministic_mode == 1: log.write(str(k) + "\t" +
525         → str(theta_minus[0]) + "\t" + str(theta_minus[1]) + "\t" +
526         → str(theta_minus[2]) + "\t" + str(theta_minus[3]) + "\t" +
527         → str(theta_minus[4]) + "\t" + str(theta_minus[5]) + "\n")
528
529     log.write("PositiveAIC: " + str(AIC_plus) + " NegativeAIC: " +
530         → str(AIC_minus) + "\n")
531
532     AIC_gradient_sample = AIC_plus - AIC_minus;
533
534     return AIC_gradient_sample
535
536 #data plotter function for monitoring SPSA results
537 def plotter(subplot,plus,minus,empirical_prob,samples,mode):
538     global plt0,plt1,plt2,plt3,plt4,plt5, prob_histo_plus,
539         → prob_histo_minus
540     bin_sequence = []
541     x_sequence = []
542     trim_value = 0
543
544     if mode == 0:
545         bin_sequence = count_bin_sequence
546         x_sequence = count_x_sequence
547         trim_value = count_trim_value
548         prob_histo_plus, bin_edges = np.histogram(plus, bin_sequence,
549             → density=True)
550         prob_histo_minus, bin_edges = np.histogram(minus, bin_sequence,
551             → density=True)
552
553     if mode == 1:
554         bin_sequence = rate_bin_sequence
555         x_sequence = rate_x_sequence
556         trim_value = rate_trim_value

```

```
544     histo_plus, bin_edges = np.histogram(plus, bin_sequence,
545         ↵ density=False)
546     histo_minus, bin_edges = np.histogram(minus, bin_sequence,
547         ↵ density=False)
548     prob_histo_plus = np.array(histo_plus / ((rate_end_seed + 1)*5))
549     prob_histo_minus = np.array(histo_plus / ((rate_end_seed + 1)*5))
550
551     trimmed_prob = empirical_prob[0:trim_value]
552
553     subplot.plot(x_sequence,trimmed_prob, 'k+')
554     plt.pause(0.0001)
555
556     interval_plus = sampler(plus,samples,bin_sequence)
557
558     subplot.plot(x_sequence,prob_histo_plus, 'g-')
559     plt.pause(0.0001)
560     subplot.fill_between(x_sequence, (prob_histo_plus - interval_plus),
561         ↵ (prob_histo_plus + interval_plus), alpha=0.2,
562         ↵ edgecolor='#008000', facecolor='#00FF00')
563     plt.pause(0.0001)
564
565     interval_minus = sampler(minus,samples,bin_sequence)
566
567     subplot.plot(x_sequence,prob_histo_minus, 'm-')
568     plt.pause(0.0001)
569     subplot.fill_between(x_sequence, (prob_histo_minus - interval_minus),
570         ↵ (prob_histo_minus + interval_minus), alpha=0.2,
571         ↵ edgecolor='#800080', facecolor='#FF00FF')
572     plt.pause(0.0001)
573
574     if subplot=plt0: plt0.set_ylim((0,.20))
575     if subplot=plt1: plt1.set_ylim(0,.20)
576     if subplot=plt2: plt2.set_ylim(0,.7)
577     if subplot=plt3: plt3.set_ylim(0, .30)
578     if subplot=plt4: plt4.set_ylim(0, .15)
579     if subplot=plt5: plt5.set_ylim(0, .35)
580
581 def sampler(data,samples,bin_sequence):
582
583     if len(data) == 0: #catches edge case w/ no entries for a mitotic
584         ↵ mode
585         data=[0]
```

```

580     base_sample=np.zeros((error_samples,len(bin_sequence)-1))
581
582     for i in range(0,error_samples):
583         new_data_sample = np.random.choice(data,samples)
584         new_histo_prob, bin_edges = np.histogram(new_data_sample,
585             ↪ bin_sequence, density=True)
586         base_sample[i,:] = new_histo_prob
587
588     sample_95CI = np.array(2 * (np.std(base_sample,0)))
589
590     return sample_95CI
591
592 # This is a helper function for run_simulation that runs bash commands in
593 # separate processes
594
595 def execute_command(cmd):
596     return subprocess.call(cmd, shell=True)
597
598 if __name__ == "__main__":
599     main()

```

15.1.9 /python_fixtures/Wan_output_fixture.py

```

1 import multiprocessing
2 import os
3 import subprocess
4 import datetime
5
6 import numpy as np
7 from openpyxl.styles.builtins import output
8 from sklearn.datasets.tests.test_svmlight_format import currdir
9
10 executable = '/home/main/chaste_build/projects/ISP/apps/WanSimulator'
11
12 if not(os.path.isfile(executable)):
13     raise Exception('Could not find executable: ' + executable)
14
15 ##########
16 # SIMULATION PARAMETERS
17 #####
18
19 #Define start and end RNG seeds; determines:
20 #No. simulated CMZs per run

```

```

21 #unique sequence of RNG results for each lineage
22 start_seed = 0
23 end_seed = 99 #total seeds should be divisible by # cores
24
25 output_directory = "WanOutput"
26
27 #####
28 #MODEL PARAMETERS
29 #####
30
31 #CMZ residency time
32 wan_residency_time = 17.0
33 #factor to divide 3dpf progenitor population by to obtain estimate of
   ↳ stem cell population
34 stem_divisor = 10
35 #3dpf CMZ progenitor population mean and standard deviation
36 progenitor_mean = 792
37 progenitor_std = 160
38
39 #stem cell cycle parameters- cell cycle RV is a shifted gamma
   ↳ distribution
40 stem_gamma_shift = 4
41 stem_gamma_shape = 6.5 #mean 30 hr stem cell time - in reality is
   ↳ probably more like 60+
42 stem_gamma_scale = 4
43
44 progenitor_gamma_shift = 4 #default He et al. values, mean 6 hr cycle
   ↳ time
45 progenitor_gamma_shape = 2
46 progenitor_gamma_scale = 1
47 progenitor_gamma_sister = 1
48
49 cmz_theta_string = str(wan_residency_time) + " " + str(stem_divisor) + "
   ↳ " + str(progenitor_mean) + " " + str(progenitor_std) + " "
   ↳ str(stem_gamma_shift) + " " + str(stem_gamma_shape) + " "
   ↳ str(stem_gamma_scale) + " " + str(progenitor_gamma_shift) + " "
   ↳ str(progenitor_gamma_shape) + " " + str(progenitor_gamma_scale) + " "
   ↳ + str(progenitor_gamma_sister)
50
51 #STOCHASTIC MITOTIC MODE
52 #HE ORIGINAL PARAMETERS
53 #mitotic mode per-phase probabilities
54 #3-phase mitotic mode time periodisation

```

```

55 mitotic_mode_phase_2 = 8 #These are phase boundaries rather than lengths
  ↵ as in eg. He_output_fixture.py
56 mitotic_mode_phase_3 = 15
57 phase_1_pPP = 1.0
58 phase_1_pPD = 0.0
59 phase_2_pPP = 0.2
60 phase_2_pPD = 0.4
61 phase_3_pPP = 0.2
62 phase_3_pPD = 0.0
63
64 stochastic_theta_string = str(mitotic_mode_phase_2) + " " +
  ↵ str(mitotic_mode_phase_3) + " " +str(phase_1_pPP) + " " +
  ↵ str(phase_1_pPD) + " " + str(phase_2_pPP) + " " + str(phase_2_pPD) +
  ↵ " " + str(phase_3_pPP) + " " + str(phase_3_pPD)
65
66 def main():
67
68     command_list = []
69
70     # Use processes equal to the number of cpus available
71     cpu_count = multiprocessing.cpu_count()
72     seeds_per_command = (end_seed + 1) / cpu_count
73     base_command = executable + " " + output_directory
74
75     for i in range(0,cpu_count):
76         curr_start_seed = i * seeds_per_command
77         curr_end_seed = curr_start_seed + (seeds_per_command - 1)
78
79         command = base_command + " "\\
80             +str(curr_start_seed) + " "\\
81             +str(curr_end_seed) + " "\\
82             +cmz_theta_string + " "\\
83             +stochastic_theta_string
84
85         #command_list.append(command)
86
87         ↵ command_list.append(" /home/main/chaste_build/projects/ISP/apps/WanSimulator
  ↵ WanOutput 18.0 24.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
  ↵ 0.2 0.4 0.2 0.0")

```

```

88
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 42.0 49.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

89
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 51.0 57.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

90
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 94.0 99.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

91
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 58.0 64.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

92
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 65.0 70.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

93
    ↳ command_list.append( "/home/main/chaste_build/projects/ISP/apps/WanSimulator
    ↳ WanOutput 71.0 74.0 17.0 10 792 160 4 6.5 4 4 2 1 1 8 15 1.0 0.0
    ↳ 0.2 0.4 0.2 0.0" )

94
print("Starting simulations with " + str(cpu_count) + " processes")

95
# Generate a pool of workers
pool = multiprocessing.Pool(processes=cpu_count)

96
97
98
99
100
# Pass the list of bash commands to the pool, block until pool is
    ↳ complete
101
pool.map(execute_command, command_list, 1)

102
103
104 # This is a helper function for run_simulation that runs bash commands in
    ↳ separate processes
105 def execute_command(cmd):
106     print("Executing command: " + cmd)
107     return subprocess.call(cmd, shell=True)

108
109
110 if __name__ == "__main__":

```

```
111     main()
```

15.1.10 /python_fixtures/diagram_utility_scripts/Gomes_He_cycle_plots.py

```
1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from fileinput import filename
5 from scipy.stats import lognorm
6 from scipy.stats import gamma
7 from statsmodels.sandbox.distributions.gof_new import a_st70_upp
8
9 #Model parameters
10 gomes_normal_sigma = .32839
11 gomes_normal_mu = 3.9716
12
13 he_gamma_shape = 2
14 he_gamma_scale = 1
15 he_gamma_offset = 4
16
17 def main():
18     gomes_x_range = np.arange(1,151,1)
19     gomes_y = lognorm.pdf(gomes_x_range, gomes_normal_sigma, 0,
20                           np.exp(gomes_normal_mu))
21
22     he_x_range = np.arange(1,16,.1)
23     he_y = gamma.pdf(he_x_range, he_gamma_shape, he_gamma_offset,
24                       he_gamma_scale)
25
26     fig, (ax_g, ax_h) = plt.subplots(1,2,figsize=(6,2))
27
28     ax_g.plot(gomes_x_range,gomes_y, 'k-')
29     ax_h.plot(he_x_range, he_y, 'k-')
30
31     plt.sca(ax_g)
32     plt.xlabel("Cell cycle duration (h)")
33     plt.ylabel("Probability")
34
35     plt.sca(ax_h)
36     plt.xlabel("Cell cycle duration (h)")
37     plt.ylabel("Probability")
```

```

36
37     plt.tight_layout()
38     plt.savefig('/home/main/Desktop/utility.png', transparent=True)
39     plt.show()
40
41 if __name__ == "__main__":
42     main()

```

15.1.11 /python_fixtures/diagram_utility_scripts/He_Boije_signal_plots.py

```

1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from fileinput import filename
5 from scipy.stats import lognorm
6 from scipy.stats import gamma
7 from statsmodels.sandbox.distributions.gof_new import a_st70_upp
8
9 #Model parameters
10 he_x = [0,8,15,25]
11 he_pp_y = [1,.2,.2,.2]
12 he_pd_y = [0,.4,0,0]
13 he_dd_y = [0,.4,.8,.8]
14
15 boije_x = [0,3,5,9]
16 boije_pAtoh7 = [0,.32,0,0]
17 boije_pPtf1a = [0,.3,0,0]
18 boije_pNg = [0,0,.8,.8]
19
20 def main():
21     fig, ((pp,pd,dd))= plt.subplots(1,3,figsize=(6,2),sharey=True)
22
23     fig2, ((pAtoh7),(pPtf1a),(pNg)) =
24         plt.subplots(3,1,figsize=(2,4),sharex=True)
25     pp.set_xlim((0,24))
26     pd.set_xlim((0,24))
27     dd.set_xlim((0,24))
28
29     pp.set_xticks([8,15])
30     pd.set_xticks([8,15])

```

```
31     dd.set_xticks([8,15])
32
33     pp.set_ylim((- .1,1.1))
34     pp.set_yticks(np.arange(0,1.2,.2))
35     pp.set_ylabel("Probability")
36     pp.set_xlabel("TiL (h)")
37     pp.axvline(8,linestyle='--',color='.1', alpha=.2)
38     pp.axvline(15,linestyle='--',color='.1', alpha=.2)
39     pp.step(he_x,he_pp_y, 'k-', where='post', linewidth=2)
40
41     pd.set_ylim((- .1,1.1))
42     pd.set_yticks(np.arange(0,1.2,.2))
43     pd.set_xlabel("TiL (h)")
44     pd.axvline(8,linestyle='--',color='.1', alpha=.2)
45     pd.axvline(15,linestyle='--',color='.1', alpha=.2)
46     pd.step(he_x,he_pd_y, 'k-', where='post', linewidth=2)
47
48     dd.set_ylim((- .1,1.1))
49     dd.set_yticks(np.arange(0,1.2,.2))
50     dd.set_xlabel("TiL (h)")
51     dd.axvline(8,linestyle='--',color='.1', alpha=.2)
52     dd.axvline(15,linestyle='--',color='.1', alpha=.2)
53     dd.step(he_x,he_dd_y, 'k-', where='post', linewidth=2)
54
55     pAtoh7.set_xlim((0,8))
56     pNg.set_xlim((0,8))
57     pPtf1a.set_xlim((0,8))
58
59     pAtoh7.set_xticks([3,5])
60     pNg.set_xticks([3,5])
61     pPtf1a.set_xticks([3,5])
62
63
64     pNg.set_xlabel("Generation")
65
66
67     pAtoh7.set_ylim((- .1,1.1))
68     pAtoh7.set_yticks(np.arange(0,1.25,.25))
69     pAtoh7.set_ylabel("Probability")
70     pAtoh7.axvline(3,linestyle='--',color='.1',alpha=.2)
71     pAtoh7.axvline(5,linestyle='--',color='.1',alpha=.2)
72     pAtoh7.step(boije_x,boije_pAtoh7, 'k-', where='post', linewidth=2)
73
```

```

74     pPtf1a.set_ylim((-1,1.1))
75     pPtf1a.set_yticks(np.arange(0,1.25,.25))
76     pPtf1a.set_ylabel("Probability")
77     pPtf1a.axvline(3,linestyle='--',color='.1',alpha=.2)
78     pPtf1a.axvline(5,linestyle='--',color='.1',alpha=.2)
79     pPtf1a.step(boije_x,boije_pPtf1a, 'k-', where='post', linewidth=2)

80
81     pNg.set_ylim((-1,1.1))
82     pNg.set_yticks(np.arange(0,1.25,.25))
83     pNg.set_ylabel("Probability")
84     pNg.axvline(3,linestyle='--',color='.1',alpha=.2)
85     pNg.axvline(5,linestyle='--',color='.1',alpha=.2)
86     pNg.step(boije_x,boije_pNg, 'k-', where='post', linewidth=2)

87
88
89     plt.tight_layout()
90     fig.savefig('Hemode.png', transparent=True)
91     fig2.savefig('Boijesignals.png', transparent=True)
92     plt.show()

93
94 if __name__ == "__main__":
95     main()

```

15.1.12 /python_fixtures/figure_plots/Cumulative_EdU.py

```

1 import os
2 import numpy as np
3 import statsmodels.api as sm
4 import pandas as pd
5 import matplotlib.pyplot as plt
6 import seaborn as sns
7
8 from PIL import Image
9 from io import BytesIO
10
11
12 #PLoS formatting stuff
13 plt.rcParams['font.size'] = 12
14 plt.rcParams['font.family'] = 'sans-serif'
15 plt.rcParams['font.sans-serif'] = ['Arial']
16
17 def main():

```

```

18 #Read & parse raw data file
19 raw_data =
20     → pd.read_excel('/home/main/git/chaste/projects/ISP/empirical_data/cumulative_e
21 #group data by dorsal/ventral
22 dv_grouped = raw_data.groupby('D/V')
23 #make new dataframe for totals and labelled fraction
24 totals_frame = dv_grouped.get_group('D')
25 #reset indices to ventral frame for addition operatinos
26 totals_frame =
27     → totals_frame.set_index(dv_grouped.get_group('V').index)
28 #sum D + V PCNA & EdU counts
29 totals_frame.PCNA = totals_frame.PCNA +
30     → dv_grouped.get_group('V').PCNA
31 totals_frame.EdU = totals_frame.EdU + dv_grouped.get_group('V').EdU
32 #create new column for labelled fraction
33 totals_frame['labelled_fraction'] = totals_frame.EdU /
34     → totals_frame.PCNA
35
36 #setup x for linear regression with y-intercept constant
37 X = totals_frame.Time
38 X = sm.add_constant(X)
39
40 model = sm.OLS(totals_frame.labelled_fraction, X).fit()
41 print(model.summary())
42 fit_results = model.params
43
44 tc = 1/fit_results[1]
45 ts = tc * fit_results[0]
46
47 sns.regplot(totals_frame.Time,totals_frame.labelled_fraction)
48
49 plt.text(7,.2,"Tc: " + f'{tc:.2f}')
50 plt.text(7,.1,"Ts: " + f'{ts:.2f}')
51
52 plt.xlabel("Time (h)")
53 plt.ylabel("Fraction of CMZ RPCs labelled by EdU")
54
55 plt.savefig("cumulative_edu.png")
56 png_memory = BytesIO()
57 plt.savefig(png_memory, format='png', dpi=600)
58 PILpng = Image.open(png_memory)
59 PILpng.save('cumulative_edu.tiff')
60 png_memory.close()

```

```
57
58     plt.show()
59
60 if __name__ == "__main__":
61     main()
```

15.1.13 /python_fixtures/figure_plots/He_output_plot.py

```
1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.patches as patches
5 from fileinput import filename
6 from statsmodels.sandbox.distributions.gof_new import a_st70_upp
7
8 from PIL import Image
9 from io import BytesIO
10
11 #PLoS formatting stuff
12 plt.rcParams['font.size'] = 10
13 plt.rcParams['font.family'] = 'sans-serif'
14 plt.rcParams['font.sans-serif'] = ['Arial']
15
16 #AIC & Plotting utility params
17 count_seeds = 10000
18 event_seeds = 1000
19 error_samples = 5000 #number of samples to draw when estimating
    ↳ plausibility interval for simulations
20
21 he_model_params = 15
22 det_model_params = 13
23
24 #stats & line_plotter utility arrays
25 bin_sequence_24_32 = np.arange(1,26,1)
26 x_sequence_24_32 = np.arange(1,25,1)
27
28 bin_sequence_48 = np.arange(1,11,1)
29 x_sequence_48 = np.arange(1,10,1)
30
31 bin_sequence_wan = np.arange(2,17,1)
32 x_sequence_wan = np.arange(2,16,1)
33
```

```

34 bin_sequence_events = np.arange(30,85,5)
35 x_sequence_events = np.arange(30,80,5)
36
37 ##########
38 # HE ET AL EMPIRICAL RESULTS
39 #########
40
41 #Count probability arrays
42 raw_counts =
    ↵ np.loadtxt('/home/main/git/chaste/projects/ISP/empirical_data/empirical_counts.cs'
    ↵ skiprows=1, usecols=(3,4,5,6,7,8,9,10)) #collect the per-cell-type
    ↵ counts
43 raw_counts_24 = raw_counts[0:64,:]
44 raw_counts_32 = raw_counts[64:233,:]
45 raw_counts_48 = raw_counts[233:396,:]
46
47 prob_empirical_24,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_24,axis=1),bin_sequence_24_32,density=True)
    ↵ #obtain probability density histogram for counts, retabulating by
    ↵ summing across types
48 prob_empirical_32,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_32,axis=1),bin_sequence_24_32,density=True)
49 prob_empirical_48,bin_edges =
    ↵ np.histogram(np.sum(raw_counts_48,axis=1),bin_sequence_48,density=True)
50
51 #no. of lineages E0 per induction timepoint/event group
52 lineages_sampled_24 = 64
53 lineages_sampled_32 = 169
54 lineages_sampled_48 = 163
55 lineages_sampled_events = 60
56
57 #Mitotic mode rate probability arrays
58 raw_events =
    ↵ np.loadtxt('/home/main/git/chaste/projects/ISP/empirical_data/empirical_lineages.cs'
    ↵ skiprows=1, usecols=(3,5,8))
59 E0_events = raw_events[np.where(raw_events[:,2]==1)] #exclude any mitosis
    ↵ whose time was too early for recording
60
61 E0_PP = E0_events[np.where(E0_events[:,0]==0)]
62 E0_PD = E0_events[np.where(E0_events[:,0]==1)]
63 E0_DD = E0_events[np.where(E0_events[:,0]==2)]
64

```

```

65 prob_empirical_PP,bin_edges =
   ↳ np.histogram(EO_PP,bin_sequence_events,density=True)
66 prob_empirical_PD,bin_edges =
   ↳ np.histogram(EO_PD,bin_sequence_events,density=True)
67 prob_empirical_DD,bin_edges =
   ↳ np.histogram(EO_DD,bin_sequence_events,density=True)

68
69 empirical_fit_list =
   ↳ [prob_empirical_24,prob_empirical_32,prob_empirical_48,prob_empirical_PP,prob_emp

70
71 #mutant results
72 wt_mean_clone_size = 6
73 ath5_mean_clone_size = 8
74 wt_even_odd_ratio = 1.1
75 ath5_even_odd_ratio = 2.2

76
77 ##########
78 # WAN ET AL EMPIRICAL RESULTS
79 #####
80 #(derived from figure)
81 prob_wan =
   ↳ [.3143,.1857,.1757,.0871,.0871,.0786,.0271,.0179,.0071,0,0,.0071];
82 lineages_sampled_wan = 40 #not reported- approximation based on 95% CI

83
84 empirical_test_list = [prob_wan, wt_mean_clone_size,
   ↳ ath5_mean_clone_size, wt_even_odd_ratio, ath5_even_odd_ratio]

85
86 def main():
87     #LOAD & PARSE HE MODEL OUTPUT FILES
88     #original fit stochastic mitotic mode files
89     o_counts_24 =
       ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
       ↳ skiprows=1, usecols=3)
90     o_counts_32 =
       ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
       ↳ skiprows=1, usecols=3)
91     o_counts_48 =
       ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
       ↳ skiprows=1, usecols=3)
92     o_events =
       ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
       ↳ skiprows=1, usecols=(0,3))
93     o_events_PP = np.array(o_events[np.where(o_events[:,1]==0)])

```

```
94     o_events_PD = np.array(o_events[np.where(o_events[:,1]==1)])
95     o_events_DD = np.array(o_events[np.where(o_events[:,1]==2)])
96     o_counts_wan_no_shadow =
97         ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
98             ↳ skiprows=1, usecols=3)
99
100    #refit stochastic mitotic mode files
101    s_counts_24 =
102        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
103            ↳ skiprows=1, usecols=3)
104    s_counts_32 =
105        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
106            ↳ skiprows=1, usecols=3)
107    s_counts_48 =
108        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
109            ↳ skiprows=1, usecols=3)
110    s_events =
111        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
112            ↳ skiprows=1, usecols=(0,3))
113    s_events_PP = np.array(s_events[np.where(s_events[:,1]==0)])
114    s_events_PD = np.array(s_events[np.where(s_events[:,1]==1)])
115    s_events_DD = np.array(s_events[np.where(s_events[:,1]==2)])
116    s_counts_wan_no_shadow =
117        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
118            ↳ skiprows=1, usecols=3)
119    s_counts_ath5 =
120        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
121            ↳ skiprows=1, usecols=3)
122    s_counts_validate =
123        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
124            ↳ skiprows=1, usecols=3)
125
126    #deterministic mitotic mode files
127    d_counts_24 =
128        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
129            ↳ skiprows=1, usecols=3)
```

```

114     d_counts_32 =
115         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
116             skiprows=1, usecols=3)
117     d_counts_48 =
118         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
119             skiprows=1, usecols=3)
120     d_events =
121         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
122             skiprows=1, usecols=(0,3))
123     d_events_PP = np.array(d_events[np.where(d_events[:,1]==0)])
124     d_events_PD = np.array(d_events[np.where(d_events[:,1]==1)])
125     d_events_DD = np.array(d_events[np.where(d_events[:,1]==2)])
126     d_counts_wan_no_shadow =
127         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
128             skiprows=1, usecols=3)
129     d_counts_ath5 =
130         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
131             skiprows=1, usecols=3)
132     d_counts_validate =
133         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
134             skiprows=1, usecols=3)

135     #calculate Ath5 clone size and even-odd ratios
136     o_wt_mean_clone_size = np.mean(o_counts_validate)
137     o_ath5_mean_clone_size = np.mean(o_counts_ath5)

138     s_wt_mean_clone_size = np.mean(s_counts_validate)
139     s_ath5_mean_clone_size = np.mean(s_counts_ath5)

140     d_wt_mean_clone_size = np.mean(d_counts_validate)
141     d_ath5_mean_clone_size = np.mean(d_counts_ath5)

142     o_wt_even_odd_ratio = len(np.where(o_counts_validate % 2 ==
143         0)[0])/len(np.where(o_counts_validate % 2 == 1)[0])
144     o_ath5_even_odd_ratio = len(np.where(o_counts_ath5 % 2 ==
145         0)[0])/len(np.where(o_counts_ath5 % 2 == 1)[0])

146     s_wt_even_odd_ratio = len(np.where(s_counts_validate % 2 ==
147         0)[0])/len(np.where(s_counts_validate % 2 == 1)[0])
148     s_ath5_even_odd_ratio = len(np.where(s_counts_ath5 % 2 ==
149         0)[0])/len(np.where(s_counts_ath5 % 2 == 1)[0])

```

```

140 d_wt_even_odd_ratio = len(np.where(d_counts_validate % 2 ==
141   ↵ 0)[0])/len(np.where(d_counts_validate % 2 == 1)[0])
d_ath5_even_odd_ratio = len(np.where(d_counts_ath5 % 2 ==
142   ↵ 0)[0])/len(np.where(d_counts_ath5 % 2 == 1)[0])

143 o_fit_output_list = [o_counts_24, o_counts_32, o_counts_48,
144   ↵ o_events_PP, o_events_PD, o_events_DD]
o_test_output_list = [o_counts_wan_no_shadow, o_wt_mean_clone_size,
145   ↵ o_ath5_mean_clone_size, o_wt_even_odd_ratio,
146   ↵ o_ath5_even_odd_ratio]

147 s_fit_output_list = [s_counts_24, s_counts_32, s_counts_48,
148   ↵ s_events_PP, s_events_PD, s_events_DD]
s_test_output_list = [s_counts_wan_no_shadow, s_wt_mean_clone_size,
149   ↵ s_ath5_mean_clone_size, s_wt_even_odd_ratio,
150   ↵ s_ath5_even_odd_ratio]

151 d_fit_output_list = [d_counts_24, d_counts_32, d_counts_48,
152   ↵ d_events_PP, d_events_PD, d_events_DD]
d_test_output_list = [d_counts_wan_no_shadow, d_wt_mean_clone_size,
153   ↵ d_ath5_mean_clone_size, d_wt_even_odd_ratio,
154   ↵ d_ath5_even_odd_ratio]

155 #SETUP FIGURES
156 #4 figures to be generated; 3 show output of individual model modes,
157   ↵ 4th overlays stochastic refit on deterministic fit
158 o_fig, ((o_24,o_32),(o_48,o_PP),(o_PD,o_DD),(o_wan_noshad,
159   ↵ o_ath5size),(o_ratio,o_unused)) =
160   ↵ plt.subplots(5,2,figsize=(7.5,8.75))
s_fig, ((s_24,s_32),(s_48,s_PP),(s_PD,s_DD),(s_wan_noshad,
161   ↵ s_ath5size),(s_ratio,s_unused)) =
162   ↵ plt.subplots(5,2,figsize=(7.5,8.75))
d_fig, ((d_24,d_32),(d_48,d_PP),(d_PD,d_DD),(d_wan_noshad,
163   ↵ d_ath5size),(d_ratio,d_unused)) =
164   ↵ plt.subplots(5,2,figsize=(7.5,8.75))
sd_fig, ((sd_24,sd_32),(sd_48,sd_PP),(sd_PD,sd_DD),(sd_wan_noshad,
165   ↵ sd_ath5size),(sd_ratio,sd_unused)) =
166   ↵ plt.subplots(5,2,figsize=(7.5,8.75))

167 #turn off unused axes
168
169   ↵ o_unused.axis('off');s_unused.axis('off');d_unused.axis('off');sd_unused.axis('off')

```

```
162 #set xlabel
163
164     → o_24.set_xlabel('Count');o_32.set_xlabel('Count');o_48.set_xlabel('Count');o_
165     → _PP.set_xlabel('Age (hpf)');o_PD.set_xlabel('Age
166     → (hpf)');o_DD.set_xlabel('Age (hpf)')
167
168
169     → s_24.set_xlabel('Count');s_32.set_xlabel('Count');s_48.set_xlabel('Count');s_
170     → _PP.set_xlabel('Age (hpf)');s_PD.set_xlabel('Age
171     → (hpf)');s_DD.set_xlabel('Age (hpf)')
172
173     → d_24.set_xlabel('Count');d_32.set_xlabel('Count');d_48.set_xlabel('Count');d_
174     → _PP.set_xlabel('Age (hpf)');d_PD.set_xlabel('Age
175     → (hpf)');d_DD.set_xlabel('Age (hpf)')
176
177 #set ylabel
178
179     → o_24.set_ylabel('Probability');o_32.set_ylabel('Probability');o_48.set_ylabel(
180     → o_PP.set_ylabel('PP Probability');o_PD.set_ylabel('PD
181     → Probability');o_DD.set_ylabel('DD Probability');
182     → o_ath5size.set_ylabel('Clone size');o_ratio.set_ylabel('Clone
183     → even/odd ratio')
184
185     → s_24.set_ylabel('Probability');s_32.set_ylabel('Probability');s_48.set_ylabel(
186     → s_PP.set_ylabel('PP Probability');s_PD.set_ylabel('PD
187     → Probability');s_DD.set_ylabel('DD Probability');
188     → s_ath5size.set_ylabel('Clone size');s_ratio.set_ylabel('Clone
189     → even/odd ratio')
190
191     → d_24.set_ylabel('Probability');d_32.set_ylabel('Probability');d_48.set_ylabel(
192     → d_PP.set_ylabel('PP Probability');d_PD.set_ylabel('PD
193     → Probability');d_DD.set_ylabel('DD Probability');
194     → d_ath5size.set_ylabel('Clone size');d_ratio.set_ylabel('Clone
195     → even/odd ratio')
```

```

188     → sd_24.set_ylabel('Probability');sd_32.set_ylabel('Probability');sd_48.set_yla
189     sd_PP.set_ylabel('PP Probability');sd_PD.set_ylabel('PD
     → Probability');sd_DD.set_ylabel('DD Probability');
190     sd_ath5size.set_ylabel('Clone size');sd_ratio.set_ylabel('Clone
     → even/odd ratio')

191 #PLOT DATA
192 #plot lines, CIs, and bars
193 o_objects = line_plotter(o_24, o_counts_24, bin_sequence_24_32,
194     → x_sequence_24_32, count_seeds, lineages_sampled_24,
195     → prob_empirical_24, 'b-', '#000080', '#0000FF', 'y+', 0)
196 line_plotter(o_32, o_counts_32, bin_sequence_24_32, x_sequence_24_32,
197     → count_seeds, lineages_sampled_32, prob_empirical_32, 'b-',
198     → '#000080', '#0000FF', 'y+', 0)
199 line_plotter(o_48, o_counts_48, bin_sequence_48, x_sequence_48,
200     → count_seeds, lineages_sampled_48, prob_empirical_48, 'b-',
201     → '#000080', '#0000FF', 'y+', 0)
202 line_plotter(o_PP, o_events_PP[:,0], bin_sequence_events,
203     → x_sequence_events, event_seeds,
204     → lineages_sampled_events,prob_empirical_PP,'b-', '#000080',
205     → '#0000FF', 'y+', 1)
206 line_plotter(o_PD, o_events_PD[:,0], bin_sequence_events,
207     → x_sequence_events, event_seeds,
208     → lineages_sampled_events,prob_empirical_PD,'b-', '#000080',
209     → '#0000FF', 'y+', 1)
210 line_plotter(o_DD, o_events_DD[:,0], bin_sequence_events,
211     → x_sequence_events, event_seeds,
212     → lineages_sampled_events,prob_empirical_DD,'b-', '#000080',
213     → '#0000FF', 'y+', 1)
214 line_plotter(o_wan_noshad, o_counts_wan_no_shadow, bin_sequence_wan,
215     → x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
216     → 'b-', '#000080', '#0000FF', 'y+', 2)
217 o_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size),
218     → (o_wt_mean_clone_size/wt_mean_clone_size),
219     → (ath5_mean_clone_size/wt_mean_clone_size),
220     → (o_ath5_mean_clone_size/wt_mean_clone_size)],color=['#000000','#000080','#000
221     → WT','OSM WT','EO Ath5mo','OSM Ath5mo'])
222 o_ratio.bar([0,1,2,3], [wt_even_odd_ratio, o_wt_even_odd_ratio,
223     → ath5_even_odd_ratio,
224     → o_ath5_even_odd_ratio],color=['#000000','#000080','#000000','#000080'],edgeco
225     → WT,'OFit WT','EO Ath5mo','OFit Ath5mo'])
226 plt.setp(o_ath5size.get_xticklabels(), rotation=25, ha='right')

```

```

204     plt.setp(o_ratio.get_xticklabels(), rotation=25, ha='right')
205
206     s_objects = line_plotter(s_24, s_counts_24, bin_sequence_24_32,
207         ↳ x_sequence_24_32, count_seeds, lineages_sampled_24,
208         ↳ prob_empirical_24, 'm-', '#800080', '#FF00FF', 'k+', 0)
209     line_plotter(s_32, s_counts_32, bin_sequence_24_32, x_sequence_24_32,
210         ↳ count_seeds, lineages_sampled_32, prob_empirical_32, 'm-',
211         ↳ '#800080', '#FF00FF', 'k+', 0)
212     line_plotter(s_48, s_counts_48, bin_sequence_48, x_sequence_48,
213         ↳ count_seeds, lineages_sampled_48, prob_empirical_48, 'm-',
214         ↳ '#800080', '#FF00FF', 'k+', 0)
215     line_plotter(s_PP, s_events_PP[:,0], bin_sequence_events,
216         ↳ x_sequence_events, event_seeds,
217         ↳ lineages_sampled_events,prob_empirical_PP,'m-', '#800080',
218         ↳ '#FF00FF', 'k+', 1)
219     line_plotter(s_PD, s_events_PD[:,0], bin_sequence_events,
220         ↳ x_sequence_events, event_seeds,
221         ↳ lineages_sampled_events,prob_empirical_PD,'m-', '#800080',
222         ↳ '#FF00FF', 'k+', 1)
223     line_plotter(s_DD, s_events_DD[:,0], bin_sequence_events,
224         ↳ x_sequence_events, event_seeds,
225         ↳ lineages_sampled_events,prob_empirical_DD,'m-', '#800080',
226         ↳ '#FF00FF', 'k+', 1)
227     line_plotter(s_wan_noshad, s_counts_wan_no_shadow, bin_sequence_wan,
228         ↳ x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
229         ↳ 'm-', '#800080', '#FF00FF', 'k+', 2)
230     s_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size),
231         ↳ (s_wt_mean_clone_size/wt_mean_clone_size),(ath5_mean_clone_size/wt_mean_clone_
232         ↳ '#000000', '#800080'],edgecolor='#000000',tick_label=['EO WT','SM
233         ↳ WT','EO Ath5mo','SM Ath5mo'])
234     s_ratio.bar([0,1,2,3], [wt_even_odd_ratio, s_wt_even_odd_ratio,
235         ↳ ath5_even_odd_ratio,
236         ↳ s_ath5_even_odd_ratio],color=['#000000','#000080','#000000','#000080'],edgeco
237         ↳ WT,'SM WT','EO Ath5mo','SM Ath5mo'])
238     plt.setp(s_ath5size.get_xticklabels(), rotation=25, ha='right')
239     plt.setp(s_ratio.get_xticklabels(), rotation=25, ha='right')

240
241     d_objects = line_plotter(d_24, d_counts_24, bin_sequence_24_32,
242         ↳ x_sequence_24_32, count_seeds, lineages_sampled_24,
243         ↳ prob_empirical_24, 'g-', '#008000', '#00FF00', 'k+', 0)

```

```

220    line_plotter(d_32, d_counts_32, bin_sequence_24_32, x_sequence_24_32,
221      ↳ count_seeds, lineages_sampled_32, prob_empirical_32, 'g-',
222      ↳ '#008000', '#00FF00', 'k+', 0)
223    line_plotter(d_48, d_counts_48, bin_sequence_48, x_sequence_48,
224      ↳ count_seeds, lineages_sampled_48, prob_empirical_48, 'g-',
225      ↳ '#008000', '#00FF00', 'k+', 0)
226    line_plotter(d_PP, d_events_PP[:,0], bin_sequence_events,
227      ↳ x_sequence_events, event_seeds,
228      ↳ lineages_sampled_events, prob_empirical_PP, 'g-', '#008000',
229      ↳ '#00FF00', 'k+', 1)
230    line_plotter(d_PD, d_events_PD[:,0], bin_sequence_events,
231      ↳ x_sequence_events, event_seeds,
232      ↳ lineages_sampled_events, prob_empirical_PD, 'g-', '#008000',
233      ↳ '#00FF00', 'k+', 1)
234    line_plotter(d_DD, d_events_DD[:,0], bin_sequence_events,
235      ↳ x_sequence_events, event_seeds,
236      ↳ lineages_sampled_events, prob_empirical_DD, 'g-', '#008000',
237      ↳ '#00FF00', 'k+', 1)
238    line_plotter(d_wan_noshad, d_counts_wan_no_shadow, bin_sequence_wan,
239      ↳ x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
240      ↳ 'g-', '#008000', '#00FF00', 'k+', 2)
241    d_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size),
242      ↳ (d_wt_mean_clone_size/wt_mean_clone_size),(ath5_mean_clone_size/wt_mean_clone_
243      ↳ WT', 'DM WT', 'EO Ath5mo', 'DM Ath5mo'])
244    d_ratio.bar([0,1,2,3], [wt_even_odd_ratio, d_wt_even_odd_ratio,
245      ↳ ath5_even_odd_ratio,
246      ↳ d_ath5_even_odd_ratio], color=['#000000', '#008000', '#000000', '#008000'], edgecolor='black')
247      ↳ WT', 'DM WT', 'EO Ath5mo', 'DM Ath5mo'])
248    plt.setp(d_ath5size.get_xticklabels(), rotation=25, ha='right')
249    plt.setp(d_ratio.get_xticklabels(), rotation=25, ha='right')
250
251
252    sd_objects_s = line_plotter(sd_24, s_counts_24, bin_sequence_24_32,
253      ↳ x_sequence_24_32, count_seeds, lineages_sampled_24,
254      ↳ prob_empirical_24, 'm-', '#800080', '#FF00FF', 'k+', 0)
255    sd_objects_d = line_plotter(sd_24, d_counts_24, bin_sequence_24_32,
256      ↳ x_sequence_24_32, count_seeds, lineages_sampled_24,
257      ↳ prob_empirical_24, 'g-', '#008000', '#00FF00', 'k+', 0)
258    line_plotter(sd_32, s_counts_32, bin_sequence_24_32,
259      ↳ x_sequence_24_32, count_seeds, lineages_sampled_32,
260      ↳ prob_empirical_32, 'm-', '#800080', '#FF00FF', 'k+', 0)

```

```

235   line_plotter(sd_32, d_counts_32, bin_sequence_24_32,
236     ↳ x_sequence_24_32, count_seeds, lineages_sampled_32,
237     ↳ prob_empirical_32, 'g-', '#008000', '#00FF00', 'k+', 0)
238   line_plotter(sd_48, s_counts_48, bin_sequence_48, x_sequence_48,
239     ↳ count_seeds, lineages_sampled_48, prob_empirical_48, 'm-',
240     ↳ '#800080', '#FF00FF', 'k+', 0)
241   line_plotter(sd_48, d_counts_48, bin_sequence_48, x_sequence_48,
242     ↳ count_seeds, lineages_sampled_48, prob_empirical_48, 'g-',
243     ↳ '#008000', '#00FF00', 'k+', 0)
244   line_plotter(sd_PP, s_events_PP[:,0], bin_sequence_events,
245     ↳ x_sequence_events, event_seeds,
246     ↳ lineages_sampled_events,prob_empirical_PP,'m-','#800080',
247     ↳ '#FF00FF', 'k+', 1)
248   line_plotter(sd_PP, d_events_PP[:,0], bin_sequence_events,
249     ↳ x_sequence_events, event_seeds,
250     ↳ lineages_sampled_events,prob_empirical_PP,'g-','#008000',
251     ↳ '#00FF00', 'k+', 1)
252   line_plotter(sd_PD, s_events_PD[:,0], bin_sequence_events,
253     ↳ x_sequence_events, event_seeds,
254     ↳ lineages_sampled_events,prob_empirical_PD,'m-','#800080',
255     ↳ '#FF00FF', 'k+', 1)
256   line_plotter(sd_PD, d_events_PD[:,0], bin_sequence_events,
257     ↳ x_sequence_events, event_seeds,
258     ↳ lineages_sampled_events,prob_empirical_PD,'g-','#008000',
259     ↳ '#00FF00', 'k+', 1)
260   line_plotter(sd_DD, s_events_DD[:,0], bin_sequence_events,
261     ↳ x_sequence_events, event_seeds,
262     ↳ lineages_sampled_events,prob_empirical_DD,'m-','#800080',
263     ↳ '#FF00FF', 'k+', 1)
264   line_plotter(sd_DD, d_events_DD[:,0], bin_sequence_events,
265     ↳ x_sequence_events, event_seeds,
266     ↳ lineages_sampled_events,prob_empirical_DD,'g-','#008000',
267     ↳ '#00FF00', 'k+', 1)
268   line_plotter(sd_wan_noshad, s_counts_wan_no_shadow, bin_sequence_wan,
269     ↳ x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
270     ↳ 'm-','#800080', '#FF00FF', 'k+', 2)
271   line_plotter(sd_wan_noshad, d_counts_wan_no_shadow, bin_sequence_wan,
272     ↳ x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
273     ↳ 'g-','#008000', '#00FF00', 'k+', 2)
274   sd_ath5size.bar([0,1,2,3,4,5], [(wt_mean_clone_size),
275     ↳ (s_wt_mean_clone_size),(d_wt_mean_clone_size),(ath5_mean_clone_size),(s_ath5_
276     ↳ WT','SM WT','DM WT', 'EO Ath5mo', 'SM Ath5Mo', 'DM Ath5mo')])

```

```

247     sd_ratio.bar([0,1,2,3,4,5], [wt_even_odd_ratio, s_wt_even_odd_ratio,
248     ↵ d_wt_even_odd_ratio, ath5_even_odd_ratio, s_ath5_even_odd_ratio,
249     ↵ d_ath5_even_odd_ratio],color=['#000000','#800080','#008000','#000000','#80008
250     ↵ WT','SM WT','DM WT', 'EO Ath5mo', 'SM Ath5Mo', 'DM Ath5mo'])
251
252     plt.setp(sd_ath5size.get_xticklabels(), rotation=25, ha='right')
253     plt.setp(sd_ratio.get_xticklabels(), rotation=25, ha='right')
254
255 #calculate AICs, write table to file
256 o_fit_AIC = calculate_fit_AIC(o_fit_output_list, empirical_fit_list,
257     ↵ he_model_params)
258 o_test_AIC = calculate_test_AIC(o_test_output_list,
259     ↵ empirical_fit_list, he_model_params)
260
261 s_fit_AIC = calculate_fit_AIC(s_fit_output_list, empirical_fit_list,
262     ↵ he_model_params)
263 s_test_AIC = calculate_test_AIC(s_test_output_list,
264     ↵ empirical_test_list, he_model_params)
265
266 d_fit_AIC = calculate_fit_AIC(d_fit_output_list, empirical_fit_list,
267     ↵ det_model_params)
268 d_test_AIC = calculate_test_AIC(d_test_output_list,
269     ↵ empirical_test_list, det_model_params)
270
271 AIC_table('AIC_table.tex', o_fit_AIC, o_test_AIC, s_fit_AIC,
272     ↵ s_test_AIC, d_fit_AIC, d_test_AIC)
273
274 #plot annotations
275 plot_annotater(s_fig,s_objects,'Stochastic Model')
276 plot_annotater(o_fig,o_objects,'Stochastic Model (original fit)')
277 plot_annotater(d_fig,d_objects, 'Deterministic Model')
278 plot_annotater(sd_fig,sd_objects_s, 'Stochastic Model', sd_objects_d,
279     ↵ 'Deterministic Model')
280
281 #export as .tiff, .png
282 o_fig.savefig('o_fig.png',dpi=600)
283 png_memory = BytesIO()
284 o_fig.savefig(png_memory, format='png', dpi=600)
285 PILpng = Image.open(png_memory)
286 PILpng.save('o_fig.tiff')
287 png_memory.close()
288
289 s_fig.savefig('s_fig.png',dpi=600)
290 png_memory = BytesIO()

```

```
279     s_fig.savefig(png_memory, format='png', dpi=600)
280     PILpng = Image.open(png_memory)
281     PILpng.save('s_fig.tiff')
282     png_memory.close()
283
284     d_fig.savefig('d_fig.png',dpi=600)
285     png_memory = BytesIO()
286     d_fig.savefig(png_memory, format='png', dpi=600)
287     PILpng = Image.open(png_memory)
288     PILpng.save('d_fig.tiff')
289     png_memory.close()
290
291     sd_fig.savefig('sd_fig.png',dpi=600)
292     png_memory = BytesIO()
293     sd_fig.savefig(png_memory, format='png', dpi=600)
294     PILpng = Image.open(png_memory)
295     PILpng.save('sd_fig.tiff')
296     png_memory.close()
297
298     plt.show()
299
300
301 def calculate_fit_AIC(output_list, empirical_list, number_params):
302     #function calculates AIC for He et al counts & event probability
303     #→ density (NOT per-lineage event probability)
304     #this numpy array holds the individual RSS vals for timepoints +
305     #→ events
306     rss = np.zeros(6)
307     number_comparisons = 0
308
309     for i in range(0,3):
310         output_counts = output_list[i]
311         empirical_prob = empirical_list[i]
312
313         output_histo,bin_edges = np.histogram(output_counts,
314             np.arange(1,len(empirical_prob)+2), density = False)
315         output_prob = np.array(output_histo/count_seeds)
316
317         residual = np.array(output_prob- empirical_prob)
318         number_comparisons += len(residual)
319         rss[i] = np.sum(np.square(residual))
320
321     for i in range (0,3):
```

```
319     output_events = output_list[i+3]
320     empirical_prob = empirical_list[i+3]
321
322     histo_events,bin_edges = np.histogram(output_events,
323         ↳ bin_sequence_events, density=True)
324     output_prob = np.array(histo_events/event_seeds)
325
326     residual = np.array(output_prob - empirical_prob)
327     number_comparisons += len(residual)
328     rss[i+3] = np.sum(np.square(residual))
329
330     AIC = 2 * number_params + number_comparisons * np.log(np.sum(rss))
331
332     return AIC
333
334 def calculate_test_AIC(output_list, empirical_list, number_params):
335     #function calculates AIC for He mutant data & Wan counts
336     rss=np.zeros(3)
337     number_comparisons = 0
338
339     #Wan data
340     for i in range (0,1):
341         output_counts = output_list[i]
342         empirical_prob = empirical_list[i]
343
344         output_histo,bin_edges = np.histogram(output_counts,
345             ↳ np.arange(2,len(empirical_prob)+3), density = False)
346         output_prob = np.array(output_histo/count_seeds)
347
348         residual = np.array(output_prob- empirical_prob)
349         number_comparisons += len(residual)
350         rss[i] = np.sum(np.square(residual))
351
352     #He data
353     for i in range(1,3):
354         output_measure = output_list[i]
355         empirical_measure = empirical_list[i]
356
357         residual = np.array(output_measure - empirical_measure)
358         number_comparisons += 1
359         rss[i] = np.sum(np.square(residual))
360
361     AIC = 2 * number_params + number_comparisons * np.log(np.sum(rss))
```

```
360
361     return AIC
362
363 #line plotter plots average counts or event event +- 2 standard
364 #    deviations (He et al's "95% probability interval")
365 # "mode" 1 gives correct hourly output for 5-hour event bins. mode 2
366 #    begins Wan plots at clone size 2, reflecting unavailable data
367 def line_plotter(subplot, data, bin_sequence, x_sequence, seeds, samples,
368 #    empirical_prob, line_colour_string, fill_edge_colour_string,
369 #    fill_face_colour_string, empirical_colour_string, mode):
370
371     legend_objects = []
372
373     prob_histo, bin_edges = np.histogram(data, bin_sequence,
374 #        density=True)
375
376     #estimate of 95% CI for empirical observations of model output from
377     #    repeated sampling of the data w/ the appropriate number of
378     #    empirically observed lineages
379     interval = sampler(data,samples,bin_sequence)
380     legend_objects.append(subplot.plot(x_sequence,prob_histo,
381 #        line_colour_string))
382     subplot.fill_between(x_sequence, (prob_histo - interval), (prob_histo
383 #        + interval), alpha=0.1, edgecolor=fill_edge_colour_string,
384 #        facecolor=fill_face_colour_string)
385
386     if mode == 0:
387
388         legend_objects.append(subplot.plot(np.arange(1,len(empirical_prob)+1),empirical_prob,
389 #            empirical_colour_string))
390
391     if mode == 1:
392
393         legend_objects.append(subplot.plot(np.arange(30,80,5),empirical_prob,
394 #            empirical_colour_string))
395
396     if mode == 2:
397
398         legend_objects.append(subplot.plot(np.arange(2,len(empirical_prob)+2,1),
399 #            empirical_prob, empirical_colour_string))
400
401     return legend_objects
402
```

```

387 def sampler(data,samples,bin_sequence):
388
389     if len(data) == 0: #catches edge case w/ no entries for a mitotic
        ↵ mode
390         data=[0]
391
392     base_sample=np.zeros((error_samples,len(bin_sequence)-1))
393
394     for i in range(0,error_samples):
395         new_data_sample = np.random.choice(data,samples)
396         new_histo_prob, bin_edges = np.histogram(new_data_sample,
            ↵ bin_sequence, density=True)
397         base_sample[i,:] = new_histo_prob
398
399     sample_95CI = np.array(2 * (np.std(base_sample,0)))
400
401     return sample_95CI
402
403 def plot_annotater(fig_to_ann, legend_objects_1, legend_objects_1_name,
    ↵ legend_objects_2=None, legend_objects_2_name=None):
404     #subplot labels
405     labels = ['A','B','C','D','E','F','G','H','I','']
406
407     for i, ax in enumerate(fig_to_ann.axes):
408         ax.text(.01,.95, labels[i], transform=ax.transAxes,
            ↵ fontweight='bold', va='top')
409
410         time_rectangle_x = .70
411         time_rectangle_y = .65
412         time_rectangle_width = .2
413         time_rectangle_height = .03
414         time_caret_half_width = .02
415         time_caret_height = np.sqrt(3)*(time_caret_half_width*2)
416
417         if i < 3:
418             #add the time bar and label 72 hr caret on appropriate plots
419
            ↵ ax.add_patch(plt.Rectangle((time_rectangle_x,time_rectangle_y),time_r
            ↵ color='k', transform=ax.transAxes))
420
            ↵ ax.add_patch(plt.Polygon(((time_rectangle_x+time_rectangle_width, time_
            ↵ fill=False, transform=ax.transAxes))

```

```
421     ↵ ax.text(time_rectangle_x+time_rectangle_width,time_rectangle_y+time_rect
        ↵ transform=ax.transAxes,
        ↵ fontsize='8',va='bottom',ha='center')

422
423 if i == 0:
424     #add 24hr caret
425
426     ↵ ax.add_patch(plt.Polygon(((time_rectangle_x,time_rectangle_y+time_rect
        ↵ fill=False, transform=ax.transAxes))

427
428 if i == 1:
429     #add 32hr caret
430
431     ↵ ax.add_patch(plt.Polygon(((time_rectangle_x+(8/48)*time_rectangle_wid
        ↵ fill=False, transform=ax.transAxes))

432
433 if i == 2:
434
435     ↵ ax.add_patch(plt.Polygon(((time_rectangle_x+(24/48)*time_rectangle_wi
        ↵ fill=False, transform=ax.transAxes))

436
437 parent_circle_coord = (.85,.8)
438 child_1_coord = (.8,.6)
439 child_2_coord = (.9,.6)
440 ellipse_width = .03
441 ellipse_height = 2.5*ellipse_width

442
443 if i == 3:
444     ↵ ax.add_patch(plt.Polygon((parent_circle_coord),child_1_coord),color=
```

```
445     ↪ ax.add_patch(plt.Polygon((parent_circle_coord),child_2_coord),color=
446     ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellipse_
     ↪ color='k', transform=ax.transAxes))
447     ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_height,
     ↪ color='k', transform=ax.transAxes))
448     ↪ ax.add_patch(patches.Ellipse(child_2_coord,ellipse_width,ellipse_height,
     ↪ color='k', transform=ax.transAxes))
449 ax.text(.91, .7, 'PP', transform=ax.transAxes, va='center',
     ↪ ha='left')
450
451 if i == 4:
452     ↪ ax.add_patch(plt.Polygon((parent_circle_coord),child_1_coord),color=
453     ↪ ax.add_patch(plt.Polygon((parent_circle_coord),child_2_coord),color=
454     ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellipse_
     ↪ height,color='k', transform=ax.transAxes))
455     ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_height,
     ↪ color='k', transform=ax.transAxes))
456     ↪ ax.add_patch(patches.Ellipse(child_2_coord,ellipse_width,ellipse_height,
     ↪ edgecolor='k', facecolor='w', transform=ax.transAxes))
457 ax.text(.91, .7, 'PD', transform=ax.transAxes, va='center',
     ↪ ha='left')
458
459
460 if i == 5:
461     ↪ ax.add_patch(plt.Polygon((parent_circle_coord),child_1_coord),color=
462     ↪ ax.add_patch(plt.Polygon((parent_circle_coord),child_2_coord),color=
463     ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellipse_
     ↪ height,color='k', transform=ax.transAxes))
464     ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_height,
     ↪ edgecolor='k', facecolor='w', transform=ax.transAxes))
```

```

465             ↵ ax.add_patch(patches.Ellipse(child_2_coord, ellipse_width, ellipse_height,
466                                         ↵ edgecolor='k', facecolor='w', transform=ax.transAxes))
467
468
469     if i == 6:
470         ax.text(.80,.85,'CMZ', transform=ax.transAxes,
471                 ↵ fontweight='bold', va='top')
472
473     if legend_objects_2 == None:
474
475         ↵ fig_to_ann.legend((legend_objects_1[1][0],legend_objects_1[0][0]),('Observer',
476                             ↵ bbox_to_anchor=(.59,.10,.1,.1), loc='upper left')
477
478     else:
479
480         ↵ fig_to_ann.legend((legend_objects_1[1][0],legend_objects_1[0][0],legend_objects_2_name),
481                             ↵ legend_objects_2_name, bbox_to_anchor=(.59,.1,.1,.1),
482                             ↵ loc='upper left')
483
484
485     #data group box annotations
486     fig_to_ann.patches.extend([plt.Rectangle((0.005,0.41),0.99,0.58,
487                                         fill=False, color='k', linestyle='-', 
488                                         transform=fig_to_ann.transFigure,
489                                         ↵ figure=fig_to_ann)])
490
491     fig_to_ann.patches.extend([plt.Polygon(((0.005, .4),(.995,.4),
492                                         ↵ (.995,.21), (.5,.21), (.5, .01), (.005, .01)),
493                                         fill=False, color='k', linestyle=':', 
494                                         transform=fig_to_ann.transFigure,
495                                         ↵ figure=fig_to_ann)])
496
497
498     fig_to_ann.text(.605, .075, 'Training Data', fontsize=12, transform =
499                     ↵ fig_to_ann.transFigure, figure=fig_to_ann)
500     fig_to_ann.patches.extend([plt.Rectangle((0.6,0.07),0.143,0.029,
501                                         fill=False, color='k', linestyle='-', 
502                                         transform=fig_to_ann.transFigure,
503                                         ↵ figure=fig_to_ann)])
504
505
506     fig_to_ann.text(.605, .025, 'Test Data', fontsize=12, transform =
507                     ↵ fig_to_ann.transFigure, figure=fig_to_ann)
508     fig_to_ann.patches.extend([plt.Rectangle((0.6,0.02),0.111,0.029,
509

```

```

493         fill=False, color='k', linestyle=':',  

494         transform=fig_to_ann.transFigure,  

495         → figure=fig_to_ann))  

496  

497     fig_to_ann.subplots_adjust(wspace=0.4, hspace=0.3)  

498     fig_to_ann.tight_layout()  

499  

500 def AIC_table (filename, o_training_AIC, o_test_AIC, s_training_AIC,  

501   → s_test_AIC, d_training_AIC, d_test_AIC):  

502     #writes LaTeX tabular snippet with AIC values  

503     table_file = open(filename, 'w')  

504     table_file.write(r'\begin{tabular}{l|l|l}'+ '\n')  

505     table_file.write(r'\hline'+ '\n')  

506     table_file.write(r'{\bf Model} & {\bf Training AIC} & {\bf Test AIC}  

507     → \\ \thickhline'+ '\n')  

508     table_file.write(r'Stochastic (He fit) &  

509     → $'+f'{o_training_AIC:.2f}'+r'$ & $' + f'{o_test_AIC:.2f}' + r'$\\  

510     → \hline'+ '\n')  

511     table_file.write(r'Stochastic (SPSA fit) &  

512     → $'+f'{s_training_AIC:.2f}'+r'$ & $' + f'{s_test_AIC:.2f}' + r'$\\  

513     → \hline'+ '\n')  

514     table_file.write(r'Deterministic (SPSA fit) &  

515     → $'+f'{d_training_AIC:.2f}'+r'$ & $' + f'{d_test_AIC:.2f}' + r'$\\  

516     → \hline'+ '\n')  

517     table_file.write(r'\end{tabular}'+ '\n')  

518  

519 if __name__ == "__main__":  

520     main()

```

15.1.14 /python_fixtures/figure_plots/Kolmogorov_plot.py

```

1 import os  

2 import numpy as np  

3 import matplotlib.pyplot as plt  

4  

5 #AIC & Plotting utility params  

6 event_seeds = 1000  

7 error_samples = 5000 #number of samples to draw when estimating  

    → plausibility interval for simulations  

8  

9 def main():  

10     #LOAD & PARSE KOLMOGOROV COMPLEXITY FILES

```

```

11     gomes_kol =
12         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/gomes_kol.txt')
13         skiprows=1, usecols=1)
14     he_kol =
15         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/he_kol.txt')
16         skiprows=1, usecols=1)
17     he_refit_kol =
18         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/he_refit_kol.txt')
19         skiprows=1, usecols=1)
20     deterministic_kol =
21         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/deterministic_kol.txt')
22         skiprows=1, usecols=1)
23     boije_kol =
24         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/boije_kol.txt')
25         skiprows=1, usecols=1)
26     eo_kol =
27         np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/KolmogorovComplexity/eo_kol.txt')
28         skiprows=1, usecols=1)

29     data_sequence =
30         [gomes_kol,he_kol,he_refit_kol,deterministic_kol,boije_kol,eo_kol]
31
32     plt.violinplot(data_sequence, showextrema=True)
33
34     plt.ylabel ('Estimated Kolmogorov Complexity')
35
36     plt.show()

37
38 if __name__ == "__main__":
39     main()

```

15.1.15 /python_fixtures/figure_plots/Mitotic_rate_plot.py

```

1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 from PIL import Image
6 from io import BytesIO
7
8 #PLoS formatting stuff
9 plt.rcParams['font.size'] = 12

```

```

10 plt.rcParams['font.family'] = 'sans-serif'
11 plt.rcParams['font.sans-serif'] = ['Arial']

12
13 #AIC & Plotting utility params
14 event_seeds = 1000
15 error_samples = 5000 #number of samples to draw when estimating
    ↳ plausibility interval for simulations

16
17 bin_sequence_events = np.arange(30,85,5)
18 x_sequence_events = np.arange(30,80,5)

19
20 ######
21 # HE ET AL EMPIRICAL RESULTS
22 #####
23
24 rate_bin_sequence = np.arange(30,85,5)
25 rate_x_sequence = np.arange(30,80,5)
26 rate_trim_value = 10

27
28 #no. of lineages E0 per induction timepoint/event group
29 lineages_sampled_events = 60

30
31 #Mitotic mode rate probability arrays
32 raw_events =
    ↳ np.loadtxt('/home/main/git/chaste/projects/ISP/empirical_data/empirical_lineages.')
    ↳ skiprows=1, usecols=(3,5,8))
33 E0_events = raw_events[np.where(raw_events[:,2]==1)] #exclude any mitosis
    ↳ whose time was too early for recording

34
35 event_counts, bin_edges =
    ↳ np.histogram(E0_events,bin_sequence_events,density=False)
36 #hourly values
37 empirical_events_per_lineage =
    ↳ np.array(event_counts/(lineages_sampled_events*5))

38
39 def main():
    #LOAD & PARSE HE MODEL OUTPUT FILES
    #original fit stochastic mitotic mode files
40     o_events =
        ↳ np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM")
        ↳ skiprows=1, usecols=(0))

41
42     #refit stochastic mitotic mode files

```

```

45     s_events =
46         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
47         skiprows=1, usecols=(0))
48
49     #deterministic mitotic mode files
50     d_events =
51         np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
52         skiprows=1, usecols=(0))
53
54     original = line_plotter(o_events,bin_sequence_events,
55         x_sequence_events, event_seeds, lineages_sampled_events, 'b--',
56         '#000080', '#0000FF')
57     refit = line_plotter(s_events,bin_sequence_events, x_sequence_events,
58         event_seeds, lineages_sampled_events, 'm-', '#800080', '#FF00FF')
59     deterministic = line_plotter(d_events,bin_sequence_events,
60         x_sequence_events, event_seeds, lineages_sampled_events, 'g-',
61         '#008000', '#00FF00')
62
63     empirical = plt.plot(np.arange(30,80,5),empirical_events_per_lineage,
64         'k+')
65
66     plt.legend((original[0], refit[0], deterministic[0], empirical[0]),
67         ('SM (He fit)', 'SM (SPSA fit)', 'DM', 'Observed'))
68
69     plt.xlabel ('Age (hpf)')
70     plt.ylabel ('Probability of mitotic event per lineage per hour')
71
72     plt.savefig("per_lineage_mitotic_rate.png")
73     png_memory = BytesIO()
74     plt.savefig(png_memory, format='png', dpi=600)
75     PILpng = Image.open(png_memory)
76     PILpng.save('per_lineage_mitotic_rate.tiff')
77     png_memory.close()
78
79     plt.show()
80
81 #line plotter plots average counts or event event +- 2 standard
82     deviations (He et al's "95% probability interval")
83 # "mode" 1 gives correct hourly output for 5-hour event bins. mode 2
84     begins Wan plots at clone size 2, reflecting unavailable data

```

```
73 def line_plotter(data, bin_sequence, x_sequence, seeds, samples,
74     ↵ line_colour_string, fill_edge_colour_string,
75     ↵ fill_face_colour_string):
76
77
78     histo, bin_edges = np.histogram(data, bin_sequence, density=False)
79     prob_histo = np.array(histo / (seeds * 5))
80
81
82     #estimate of 95% CI for empirical observations of model output from
83     ↵ repeated sampling of the data w/ the appropriate number of
84     ↵ empirically observed lineages
85     interval = sampler(data,samples,bin_sequence)
86     line = plt.plot(x_sequence,prob_histo, line_colour_string)
87     plt.fill_between(x_sequence, (prob_histo - interval), (prob_histo +
88         ↵ interval), alpha=0.1, edgecolor=fill_edge_colour_string,
89         ↵ facecolor=fill_face_colour_string)
90
91
92     return line
93
94
95 def sampler(data,samples,bin_sequence):
96
97     if len(data) == 0: #catches edge case w/ no entries for a mitotic
98         ↵ mode
99         data=[0]
100
101
102     base_sample=np.zeros((error_samples,len(bin_sequence)-1))
103
104
105     for i in range(0,error_samples):
106         new_data_sample = np.random.choice(data,samples)
107         new_histo, bin_edges = np.histogram(new_data_sample,
108             ↵ bin_sequence, density=False)
109         new_histo_prob = np.array(new_histo/samples)
110         base_sample[i,:] = new_histo_prob
111
112
113     sample_95CI = np.array(2 * (np.std(base_sample,0)))
114
115
116     return sample_95CI
117
118
119 if __name__ == "__main__":
120     main()
```

15.1.16 /python_fixtures/figure_plots/Wan_output_plot.py

```

1 import os
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import matplotlib.ticker as ticker
6
7 from PIL import Image
8 from io import BytesIO
9
10 #PLoS formatting stuff
11 plt.rcParams['font.size'] = 12
12 plt.rcParams['font.family'] = 'sans-serif'
13 plt.rcParams['font.sans-serif'] = ['Arial']
14
15 def main():
16     wan_output_dir =
17         '/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/WanOutput'
18     wan_results_list = []
19     for root,dirs,files in os.walk(wan_output_dir):
20         for name in files:
21             if name == 'celltypes.dat':
22                 results = np.loadtxt(os.path.join(root,name), usecols=2)
23                 if len(results) == 8569:
24                     wan_results_list.append(results)
25
26     wan_sim_results = np.array(wan_results_list)
27     wan_sim_mean = np.mean(wan_sim_results, axis=0)
28     wan_sim_95CI = 2*np.std(wan_sim_results, axis=0)
29     wan_sim_x_seq = np.arange(72,8641,1)/24
30
31     empirical_pop_mean = np.array([792.0, 768.4, 906.1, 1159.7, 1630.0,
32         ↵ 3157.9, 3480.2, 4105.1, 1003.0, 477.2, 438.8088611111])
33     empirical_pop_95CI = 2*np.array([160.1, 200.1, 244.5, 477.6, 444.3,
34         ↵ 1414.3, 472.1, 1169.7, 422.8, 367.5, 294.8])
35     empirical_x_seq =
36         ↵ np.array([72,120,192,288,408,552,720,1440,2160,4320,8640])/24
37
38     fig, ax = plt.subplots(1,1)
39
40     empirical = plt.plot(empirical_x_seq, empirical_pop_mean, 'g-')

```

```

37     plt.fill_between(empirical_x_seq, (empirical_pop_mean -
38         ↵ empirical_pop_95CI), (empirical_pop_mean + empirical_pop_95CI),
39         ↵ alpha=0.1, edgecolor='#008000', facecolor='#00FF00')
40
41     wan_simulator = plt.plot(wan_sim_x_seq, wan_sim_mean, 'm-')
42     plt.fill_between(wan_sim_x_seq, (wan_sim_mean - wan_sim_95CI),
43         ↵ (wan_sim_mean + wan_sim_95CI), alpha=0.1, edgecolor='#800080',
44         ↵ facecolor='#FF00FF')
45
46     plt.ylim(bottom=0)
47     plt.xlim(left=0)
48
49     plt.ylabel("CMZ population size")
50     plt.xlabel("Retina age (dpf)")
51     ax.xaxis.set_major_locator(ticker.MultipleLocator(40))
52     ax.legend((empirical[0], wan_simulator[0]), ('Observed CMZ
53         ↵ population', '"Wan-type" simulated CMZ population'))
54
55     png_memory = BytesIO()
56     fig.savefig(png_memory, format='png', dpi=600)
57     PILpng = Image.open(png_memory)
58     PILpng.save('wan_fig.tiff')
59     png_memory.close()
60
61     plt.show()
62
63
64 if __name__ == "__main__":
65     main()

```

15.1.17 /src/BoijeCellCycleModel.cpp

```

1 #include "BoijeCellCycleModel.hpp"
2
3 BoijeCellCycleModel::BoijeCellCycleModel() :
4     AbstractSimpleCellCycleModel(), mOutput(false),
5     ↵ mEventStartTime(), mSequenceSampler(false),
6     ↵ mSeqSamplerLabelSister(
7         false), mDebug(false), mTimeID(), mVarIDs(),
8         ↵ mDebugWriter(), mGeneration(0), mPhase2gen(3),
9         ↵ mPhase3gen(
10            5), mprobAtoh7(0.32), mprobPtf1a(0.30), mprobng(0.80),
11            ↵ mAtoh7Signal(false), mPtf1aSignal(false), mNgSignal(

```

```

7             false), mMitoticMode(0), mSeed(0), mp_PostMitoticType(),
8     ↳ mp_RGC_Type(), mp_AC_HC_Type(), mp_PR_BC_Type(),
9     ↳ mp_label_Type()
10    }
11
12 BoijeCellCycleModel::BoijeCellCycleModel(const BoijeCellCycleModel&
13   ↳ rModel) :
14     AbstractSimpleCellCycleModel(rModel), mOutput(rModel.mOutput),
15     ↳ mEventStartTime(rModel.mEventStartTime), mSequenceSampler(
16       rModel.mSequenceSampler),
17       ↳ mSeqSamplerLabelSister(rModel.mSeqSamplerLabelSister),
18       ↳ mDebug(rModel.mDebug), mTimeID(
19         rModel.mTimeID), mVarIDs(rModel.mVarIDs),
20         ↳ mDebugWriter(rModel.mDebugWriter), mGeneration(
21           rModel.mGeneration), mPhase2gen(rModel.mPhase2gen),
22           ↳ mPhase3gen(rModel.mPhase3gen), mprobAtoh7(
23             rModel.mprobAtoh7), mprobPtf1a(rModel.mprobPtf1a),
24             ↳ mprobng(rModel.mprobng), mAtoh7Signal(
25               rModel.mAtoh7Signal), mPtf1aSignal(rModel.mPtf1aSignal),
26               ↳ mNgSignal(rModel.mNgSignal), mMitoticMode(
27                 rModel.mMitoticMode), mSeed(rModel.mSeed),
28                 ↳ mp_PostMitoticType(rModel.mp_PostMitoticType),
29                 ↳ mp_RGC_Type(
30                   rModel.mp_RGC_Type), mp_AC_HC_Type(rModel.mp_AC_HC_Type),
31                   ↳ mp_PR_BC_Type(rModel.mp_PR_BC_Type), mp_label_Type(
32                     rModel.mp_label_Type)
33
34 {
35 }
```

23

```

24 AbstractCellCycleModel* BoijeCellCycleModel::CreateCellCycleModel()
25 {
26     return new BoijeCellCycleModel(*this);
27 }
```

28

```

29 void BoijeCellCycleModel::SetCellCycleDuration()
30 {
31
32     if
33     ↳ (mpCell->GetCellProliferativeType()->IsType<DifferentiatedCellProliferativeTy
34     {
35         mCellCycleDuration = DBL_MAX;
36     }
```

```

36     else
37     {
38         mCellCycleDuration = 1.0;
39     }
40 }
41 }
42
43 void BoijeCellCycleModel::ResetForDivision()
44 {
45     mGeneration++; //increment generation counter
46     //the first division is ascribed to generation "1"
47
48     RandomNumberGenerator* p_random_number_generator =
49     → RandomNumberGenerator::Instance();
50
51     mMitoticMode = 0; //0=PP;1=PD;2=DD
52
53     /*****
54      * TRANSCRIPTION FACTOR RANDOM VARIABLES & RULES
55      * 1st phase: only PP divisions (symmetric proliferative) permitted
56      * 2nd phase: all division modes permitted, all TF signals avail
57      * 3rd phase: only PP/DD modes permitted, only ng has nonzero signal
58      *****/
59
60     //reset RVs and signals
61     double atoh7RV = 1, ptf1aRV = 1, ngRV = 1;
62     mAtoh7Signal = mPtf1aSignal = mNgSignal = false;
63
64     //PHASE & TF SIGNAL RULES
65     if (mGeneration > mPhase2gen && mGeneration ≤ mPhase3gen) //if the
66     → cell is in the 2nd model phase, all signals have nonzero
67     → probabilities at each division
68     {
69         //RVs take values evenly distributed across 0-1
70         atoh7RV = p_random_number_generator→ranf();
71         ptf1aRV = p_random_number_generator→ranf();
72         ngRV = p_random_number_generator→ranf();
73
74         if (atoh7RV < mprobAtoh7)
75         {
76             mAtoh7Signal = true;
77         }

```

```

76     if (ptf1aRV < mprobPtf1a)
77     {
78         mPtf1aSignal = true;
79     }
80     if (ngRV < mprobng)
81     {
82         mNgSignal = true;
83     }
84 }
85
86 if (mGeneration > mPhase3gen) //if the cell is in the 3rd model
87   ↪ phase, only ng signal has a nonzero probability
88 {
89     ngRV = p_random_number_generator→ranf();
90     //roll a probability die for the ng signal
91     if (ngRV < mprobng)
92     {
93         mNgSignal = true;
94     }
95
96 /*****
97 * MITOTIC MODE & SPECIFICATION RULES
98 * -(additional asymmetric postmitotic rules specified in
99 ← InitialiseDaughterCell());
100 * *****/
101
102 if(mAtoh7Signal == true)
103 {
104     mMitoticMode = 1;
105 }
106
107 if (mPtf1aSignal == true && mAtoh7Signal == false) //Ptf1A alone
108   ↪ gives a symmetrical postmitotic AC/HC division
109 {
110     mMitoticMode = 2;
111     mpCell→SetCellProliferativeType(mp_PostMitoticType);
112     mpCell→AddCellProperty(mp_AC_HC_Type);
113 }
114
115 if (mPtf1aSignal == false && mAtoh7Signal == false && mNgSignal ==
116   ↪ true) //ng alone gives a symmetrical postmitotic PR/BC division
117 {

```

```
115     mMitoticMode = 2;
116     mpCell->SetCellProliferativeType(mp_PostMitoticType);
117     mpCell->AddCellProperty(mp_PR_BC_Type);
118 }
119
120 /*****
121 * Write mitotic event to file if appropriate
122 * mDebug: many files: detailed per-lineage info switch; intended for
123 * TestHeInductionCountFixture
124 * mOutput: 1 file: time, seed, cellID, mitotic mode, intended for
125 * TestHeMitoticModeRateFixture
126 * *****/
127
128 if (mDebug)
129 {
130     WriteDebugData(atoh7RV, ptf1aRV, ngRV);
131 }
132
133 if (mOutput)
134 {
135     WriteModeEventOutput();
136 }
137
138 /*****
139 * SEQUENCE SAMPLER
140 *****/
141 //if the sequence sampler has been turned on, check for the label &
142 // write mitotic mode to log
143 //50% chance of each daughter cell from a mitosis inheriting the
144 // label
145 if (mSequenceSampler)
146 {
147     if (mpCell->HasCellProperty<CellLabel>())
148     {
149         (*LogFile::Instance()) << mMitoticMode;
150         double labelRV = p_random_number_generator->ranf();
151         if (labelRV <= .5)
152         {
153             mSeqSamplerLabelSister = true;
154             mpCell->RemoveCellProperty<CellLabel>();
155         }
156     }
157 }
```

```

154         else
155     {
156         mSeqSamplerLabelSister = false;
157     }
158 }
159 else
160 {
161     //prevents lost-label cells from labelling their progeny
162     mSeqSamplerLabelSister = false;
163 }
164 }
165 }
166
167 void BoijeCellCycleModel :: InitialiseDaughterCell()
168 {
169     //Asymmetric specification rules
170
171     if (mAtoh7Signal = true)
172     {
173         if (mPtf1aSignal = true)
174         {
175             mpCell->SetCellProliferativeType(mp_PostMitoticType);
176             mpCell->AddCellProperty(mp_AC_HC_Type);
177         }
178         else
179         {
180             mpCell->SetCellProliferativeType(mp_PostMitoticType);
181             mpCell->AddCellProperty(mp_RGC_Type);
182         }
183     }
184
185     /*****
186     * SEQUENCE SAMPLER
187     *****/
188     if (mSequenceSampler)
189     {
190         if (mSeqSamplerLabelSister)
191         {
192             mpCell->AddCellProperty(mp_label_Type);
193             mSeqSamplerLabelSister = false;
194         }
195         else
196         {

```

```
197         mpCell->RemoveCellProperty<CellLabel>();
198     }
199 }
200 }
201
202 void BoijeCellCycleModel::SetGeneration(unsigned generation)
203 {
204     mGeneration = generation;
205 }
206
207 unsigned BoijeCellCycleModel::GetGeneration() const
208 {
209     return mGeneration;
210 }
211
212 void
213     ← BoijeCellCycleModel::SetPostMitoticType(boost::shared_ptr<AbstractCellProperty>
214     ← p_PostMitoticType)
215 {
216     mp_PostMitoticType = p_PostMitoticType;
217
218     ← boost::shared_ptr<AbstractCellProperty>
219     ← p_AC_HC_Type,
220
221     ← boost::shared_ptr<AbstractCellProperty>
222     ← p_PR_BC_Type)
223 {
224
225     mp_RGC_Type = p_RGC_Type;
226     mp_AC_HC_Type = p_AC_HC_Type;
227     mp_PR_BC_Type = p_PR_BC_Type;
228 }
229
230
231 void BoijeCellCycleModel::SetModelParameters(unsigned phase2gen, unsigned
232     ← phase3gen, double probAtoh7, double probPtf1a,
233                                     double probng)
234 {
235
236     mPhase2gen = phase2gen;
237     mPhase3gen = phase3gen;
238     mprobAtoh7 = probAtoh7;
```

```
231     mprobPtf1a = probPtf1a;
232     mprobng = probng;
233 }
234
235 void BoijeCellCycleModel::EnableModeEventOutput(double eventStart,
236     ↪ unsigned seed)
237 {
238     mOutput = true;
239     mEventStartTime = eventStart;
240     mSeed = seed;
241 }
242
243 void BoijeCellCycleModel::WriteModeEventOutput()
244 {
245     double currentTime = SimulationTime::Instance()→GetTime() +
246         ↪ mEventStartTime;
247     CellPtr currentCell = GetCell();
248     double currentCellID = (double) currentCell→GetCellId();
249     (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" <<
250         ↪ currentCellID << "\t" << mMitoticMode << "\n";
251 }
252
253 void
254     ↪ BoijeCellCycleModel::EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty>
255         ↪ label)
256 {
257     mSequenceSampler = true;
258     mp_label_Type = label;
259 }
260
261
262 void
263     ↪ BoijeCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
264         ↪ debugWriter)
265 {
266     mDebug = true;
267     mDebugWriter = debugWriter;
268
269     mTimeID = mDebugWriter→DefineUnlimitedDimension("Time", "h");
270     mVarIDs.push_back(mDebugWriter→DefineVariable("CellID", "No"));
271     mVarIDs.push_back(mDebugWriter→DefineVariable("Generation", "No"));
272     mVarIDs.push_back(mDebugWriter→DefineVariable("MitoticMode",
273         ↪ "Mode"));
```

```

266     mVarIDs.push_back(mDebugWriter->DefineVariable("atoh7Set",
267         "Percentile"));
268     mVarIDs.push_back(mDebugWriter->DefineVariable("atoh7RV",
269         "Percentile"));
270     mVarIDs.push_back(mDebugWriter->DefineVariable("ptf1aSet",
271         "Percentile"));
272     mVarIDs.push_back(mDebugWriter->DefineVariable("ptf1aRV",
273         "Percentile"));
274     mVarIDs.push_back(mDebugWriter->DefineVariable("ngSet",
275         "Percentile"));
276     mVarIDs.push_back(mDebugWriter->DefineVariable("ngRV",
277         "Percentile"));

278     mDebugWriter->EndDefineMode();
279 }

280 void BoijeCellCycleModel::WriteDebugData(double atoh7RV, double ptf1aRV,
281     double ngRV)
282 {
283     double currentTime = SimulationTime::Instance()->GetTime();
284     CellPtr currentCell = GetCell();
285     double currentCellID = (double) currentCell->GetCellId();

286     mDebugWriter->PutVariable(mTimeID, currentTime);
287     mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
288     mDebugWriter->PutVariable(mVarIDs[1], mGeneration);
289     mDebugWriter->PutVariable(mVarIDs[2], mMitoticMode);
290     mDebugWriter->PutVariable(mVarIDs[3], mprobAtoh7);
291     mDebugWriter->PutVariable(mVarIDs[4], atoh7RV);
292     mDebugWriter->PutVariable(mVarIDs[5], mprobPtf1a);
293     mDebugWriter->PutVariable(mVarIDs[6], ptf1aRV);
294     mDebugWriter->PutVariable(mVarIDs[7], mprobng);
295     mDebugWriter->PutVariable(mVarIDs[8], ngRV);
296     mDebugWriter->AdvanceAlongUnlimitedDimension();
297 }

298 ****
299 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
300 ****/
301
302 double BoijeCellCycleModel::GetAverageTransitCellCycleTime()
303 {
304     return mCellCycleDuration;

```

```

302 }
303
304 double BoijeCellCycleModel::GetAverageStemCellCycleTime()
305 {
306     return mCellCycleDuration;
307 }
308
309 void BoijeCellCycleModel::OutputCellCycleModelParameters(out_stream&
310             rParamsFile)
311 {
312     *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
313             "</CellCycleDuration>\n";
314
315     // Call method on direct parent class
316
317     ↵ AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
318
319 // Serialization for Boost ≥ 1.36
320 #include "SerializationExportWrapperForCpp.hpp"
321 CHASTE_CLASS_EXPORT(BoijeCellCycleModel)

```

15.1.18 /src/BoijeCellCycleModel.hpp

```

1 #ifndef BOIJECELLCYCLEMODEL_HPP_
2 #define BOIJECELLCYCLEMODEL_HPP_
3
4 #include "AbstractSimpleCellCycleModel.hpp"
5 #include "RandomNumberGenerator.hpp"
6 #include "Cell.hpp"
7 #include "DifferentiatedCellProliferativeType.hpp"
8 #include "SmartPointers.hpp"
9 #include "ColumnDataWriter.hpp"
10 #include "LogFile.hpp"
11 #include "CellLabel.hpp"
12
13 #include "BoijeRetinalNeuralFates.hpp"
14
15
16 /*****
17 * BOIJE CELL CYCLE MODEL

```

```

18 * As described in Boije et al. 2015 [Boije2015] doi:
→ 10.1016/j.devcel.2015.08.011
19 *
20 * USE: By default, BoijeCellCycleModels are constructed with the
→ parameters reported in [Boije2015].
21 * In normal use, the model steps through three phases of probabilistic
→ transcription factor signalling.
22 * These signals determine the mitotic mode and the fate of offspring.
23 *
24 * NB: the Boije model is purely generational and assumes a generation
→ time of 1; time in Boije simulations
25 * thus represents generation number rather than proper time. Refactor for
→ use with simulations that refer
26 * to clocktime!!
27 *
28 * 2 per-model-event output modes:
29 * EnableModeEventOutput() enables mitotic mode event logging-all cells
→ will write to the singleton log file
30 * EnableModelDebugOutput() enables more detailed debug output, each seed
→ will have its own file written to
31 * by a ColumnDataWriter passed to it from the test
32 * (eg. by the SetupDebugOutput helper function in the project simulator)
33 *
34 * 1 mitotic-event-sequence sampler (only samples one "path" through the
→ lineage):
35 * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic
→ event type to a string in the singleton log file
36 *
37 *****/
38
39 class BoijeCellCycleModel : public AbstractSimpleCellCycleModel
40 {
41     friend class TestSimpleCellCycleModels;
42
43 private:
44
45     /** Needed for serialization.*/
46     friend class boost::serialization::access;
47     /**
48      * Archive the cell-cycle model and random number generator, never
→ used directly - boost uses this.
49      *
50      * @param archive the archive

```

```
51     * @param version the current version of this class
52     */
53     template<class Archive>
54     void serialize(Archive & archive, const unsigned int version)
55     {
56         archive &
57             → boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);
58
59         SerializableSingleton<RandomNumberGenerator>* p_wrapper =
60             → RandomNumberGenerator::Instance()→GetSerializationWrapper();
61         archive & p_wrapper;
62         archive & mCellCycleDuration;
63         archive & mGeneration;
64     }
65
66     //Private write functions for models
67     void WriteModeEventOutput();
68     void WriteDebugData(double atoh7RV, double ptf1aRV, double ngRV);
69
70 protected:
71     //mode/output variables
72     bool mOutput;
73     double mEventStartTime;
74     bool mSequenceSampler;
75     bool mSeqSamplerLabelSister;
76     //debug writer stuff
77     bool mDebug;
78     int mTimeID;
79     std::vector<int> mVarIDs;
80     boost::shared_ptr<ColumnDataWriter> mDebugWriter;
81     //model parameters and state memory vars
82     unsigned mGeneration;
83     unsigned mPhase2gen;
84     unsigned mPhase3gen;
85     double mprobAtoh7;
86         double mprobPtf1a;
87         double mprobng;
88         bool mAtoh7Signal;
89         bool mPtf1aSignal;
90         bool mNgSignal;
91     unsigned mMitoticMode;
92     unsigned mSeed;
93     boost::shared_ptr<AbstractCellProperty> mp_PostMitoticType;
```

```
92     boost::shared_ptr<AbstractCellProperty> mp_RGC_Type;
93     boost::shared_ptr<AbstractCellProperty> mp_AC_HC_Type;
94     boost::shared_ptr<AbstractCellProperty> mp_PR_BC_Type;
95     boost::shared_ptr<AbstractCellProperty> mp_label_Type;
96
97     /**
98      * Protected copy-constructor for use by CreateCellCycleModel().
99      *
100     * The only way for external code to create a copy of a cell cycle
101    ← model
102    ← * is by calling that method, to ensure that a model of the correct
103    ← subclass is created.
104    ← * This copy-constructor helps subclasses to ensure that all member
105    ← variables are correctly copied when this happens.
106    ← *
107    ← * This method is called by child classes to set member variables for
108    ← a daughter cell upon cell division.
109    ← * Note that the parent cell cycle model will have had
110    ← ResetForDivision() called just before CreateCellCycleModel() is
111    ← called,
112    ← * so performing an exact copy of the parent is suitable behaviour.
113    ← Any daughter-cell-specific initialisation
114    ← * can be done in InitialiseDaughterCell().
115    ← *
116    ← * @param rModel the cell cycle model to copy.
117    ← */
118 BoijeCellCycleModel(const BoijeCellCycleModel& rModel);
119
120 public:
121
122 /**
123  * Constructor - just a default, mBirthTime is set in the
124  ← AbstractCellCycleModel class.
125  */
126 BoijeCellCycleModel();
127
128 /**
129  * SetCellCycleDuration() method to set length of cell cycle (default
130  ← 1.0 as Boije's model is purely generational)
131  */
132 void SetCellCycleDuration();
133
134 /**
135  *
```

```
126     * Overridden builder method to create new copies of
127     * this cell-cycle model.
128     *
129     * @return new cell-cycle model
130     */
131 AbstractCellCycleModel* CreateCellCycleModel();
132
133 /**
134  * Overridden ResetForDivision() method.
135  * Contains general mitotic mode logic
136  */
137 void ResetForDivision();
138
139 /**
140  * Overridden InitialiseDaughterCell() method.
141  * Used to implement asymmetric mitotic mode
142  */
143 void InitialiseDaughterCell();
144
145 /**
146  * Sets the cell's generation.
147  *
148  * @param generation the cell's generation
149  */
150 void SetGeneration(unsigned generation);
151
152 /**
153  * @return the cell's generation.
154  */
155 unsigned GetGeneration() const;
156
157 ****
158  * Model setup functions:
159  * Set TF signal probabilities with SetModelParameters
160  * Set Postmitotic and specification AbstractCellProperties w/ the
161  * relevant functions
162  * By default these are available in BoijeRetinalNeuralFates.hpp
163  *
164  * NB: The Boije model lumps together amacrine & horizontal cells,
165  * photoreceptors & bipolar neurons
166 ****/
```

```

166     void SetModelParameters(unsigned phase2gen = 3, unsigned phase3gen =
167     ↵ 5, double probAtoh7 = 0.32, double probPtf1a = 0.3, double probng
168     ↵ = 0.8);
169
170     void SetPostMitoticType(boost::shared_ptr<AbstractCellProperty>
171     ↵ p_PostMitoticType);
172
173     void SetSpecifiedTypes(boost::shared_ptr<AbstractCellProperty>
174     ↵ p_RGC_Type, boost::shared_ptr<AbstractCellProperty> p_AC_HC_Type,
175     ↵ boost::shared_ptr<AbstractCellProperty> p_PR_BC_Type);
176
177
178 //Functions to enable per-cell mitotic mode logging for mode rate &
179 //sequence sampling fixtures
180 //Uses singleton logfile
181
182     void EnableModeEventOutput(double eventStart, unsigned seed);
183
184     void EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty>
185     ↵ label);
186
187
188 //More detailed debug output. Needs a ColumnDataWriter passed to it
189 //Only declare ColumnDataWriter directory, filename, etc; do not set
190 //up otherwise
191
192     void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
193     ↵ debugWriter);
194
195
196 /**
197 * Overridden GetAverageTransitCellCycleTime() method.
198 *
199 * @return the average of mMinCellCycleDuration and
200 mMaxCellCycleDuration
201 */
202
203 double GetAverageTransitCellCycleTime();
204
205
206 /**
207 * Overridden GetAverageStemCellCycleTime() method.
208 *
209 * @return the average of mMinCellCycleDuration and
210 mMaxCellCycleDuration
211 */
212
213 double GetAverageStemCellCycleTime();
214
215
216 /**
217 * Overridden OutputCellCycleModelParameters() method.
218 *
219 * @param rParamsFile the file stream to which the parameters are
220 output

```

```

197     */
198     virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
199 };
200
201 #include "SerializationExportWrapper.hpp"
202 // Declare identifier for the serializer
203 CHASTE_CLASS_EXPORT(BoijeCellCycleModel)
204
205 #endif /*BOIJECELLCYCLEMODEL_HPP_*/

```

15.1.19 /src/BoijeRetinalNeuralFates.cpp

```

1 #include " ../../../../../projects/ISP/src/BoijeRetinalNeuralFates.hpp"
2
3 RetinalGanglion::RetinalGanglion(unsigned colour)
4     : AbstractCellProperty(),
5      mColour(colour)
6 {
7 }
8
9 RetinalGanglion::~RetinalGanglion()
10 {
11 }
12
13 unsigned RetinalGanglion::GetColour() const
14 {
15     return mColour;
16 }
17
18 AmacrineHorizontal::AmacrineHorizontal(unsigned colour)
19     : AbstractCellProperty(),
20      mColour(colour)
21 {
22 }
23
24 AmacrineHorizontal::~AmacrineHorizontal()
25 {
26 }
27
28 unsigned AmacrineHorizontal::GetColour() const
29 {
30     return mColour;

```

```

31 }
32
33 ReceptorBipolar::ReceptorBipolar(unsigned colour)
34     : AbstractCellProperty(),
35     mColour(colour)
36 {
37 }
38
39 ReceptorBipolar::~ReceptorBipolar()
40 {
41 }
42
43 unsigned ReceptorBipolar::GetColour() const
44 {
45     return mColour;
46 }
47
48 #include "SerializationExportWrapperForCpp.hpp"
49 // Declare identifier for the serializer
50 CHASTE_CLASS_EXPORT(RetinalGanglion)
51 CHASTE_CLASS_EXPORT(AmacrineHorizontal)
52 CHASTE_CLASS_EXPORT(ReceptorBipolar)

```

15.1.20 /src/BoijeRetinalNeuralFates.hpp

```

1 #ifndef BOIJERETINALNEURALFATES_HPP_
2 #define BOIJERETINALNEURALFATES_HPP_
3
4 #include <boost/shared_ptr.hpp>
5 #include "AbstractCellProperty.hpp"
6 #include "ChasteSerialization.hpp"
7 #include <boost/serialization/base_object.hpp>
8
9 class RetinalGanglion : public AbstractCellProperty
10 {
11 private:
12
13     /**
14      * Colour for use by visualizer.
15      */
16     unsigned mColour;
17

```

```
18  /** Needed for serialization.*/
19  friend class boost::serialization::access;
20  /**
21   * Archive the member variables.
22   *
23   * @param archive the archive
24   * @param version the current version of this class
25   */
26  template<class Archive>
27  void serialize(Archive & archive, const unsigned int version)
28  {
29      archive &
30          boost::serialization::base_object<AbstractCellProperty>(*this);
31      archive & mColour;
32  }
33
34 public:
35 /**
36  * Constructor.
37  *
38  * @param colour what colour cells with this property should be in
39  * the visualizer (defaults to 6)
40  */
41 RetinalGanglion(unsigned colour=3);
42 /**
43  * Destructor.
44  */
45 virtual ~RetinalGanglion();
46
47 /**
48  * @return #mColour.
49  */
50 unsigned GetColour() const;
51 };
52
53 class AmacrineHorizontal : public AbstractCellProperty
54 {
55 private:
56 /**
57  * Colour for use by visualizer.
```

```
59     */
60     unsigned mColour;
61
62     /** Needed for serialization.*/
63     friend class boost::serialization::access;
64     /**
65      * Archive the member variables.
66      *
67      * @param archive the archive
68      * @param version the current version of this class
69      */
70     template<class Archive>
71     void serialize(Archive & archive, const unsigned int version)
72     {
73         archive &
74             boost::serialization::base_object<AbstractCellProperty>(*this);
75         archive & mColour;
76     }
77
78 public:
79 /**
80  * Constructor.
81  *
82  * @param colour what colour cells with this property should be in
83  * the visualizer (defaults to 6)
84  */
85     AmacrineHorizontal(unsigned colour=4);
86
87 /**
88  * Destructor.
89  */
90     virtual ~AmacrineHorizontal();
91
92 /**
93  * @return #mColour.
94  */
95     unsigned GetColour() const;
96 };
97
98 class ReceptorBipolar : public AbstractCellProperty
99 {
```

```
100
101  /**
102   * Colour for use by visualizer.
103   */
104  unsigned mColour;
105
106  /** Needed for serialization. */
107  friend class boost::serialization::access;
108  /**
109   * Archive the member variables.
110   *
111   * @param archive the archive
112   * @param version the current version of this class
113   */
114  template<class Archive>
115  void serialize(Archive & archive, const unsigned int version)
116  {
117      archive &
118          → boost::serialization::base_object<AbstractCellProperty>(*this);
119      archive & mColour;
120  }
121
122
123  /**
124   * Constructor.
125   *
126   * @param colour what colour cells with this property should be in
127   * the visualizer (defaults to 6)
128   */
129  ReceptorBipolar(unsigned colour=5);
130
131  /**
132   * Destructor.
133   */
134  virtual ~ReceptorBipolar();
135
136  /**
137   * @return #mColour.
138   */
139  unsigned GetColour() const;
140};
```

```
141 #include "SerializationExportWrapper.hpp"  
142 // Declare identifier for the serializer  
143 CHASTE_CLASS_EXPORT(RetinalGanglion)  
144 CHASTE_CLASS_EXPORT(amacrineHorizontal)  
145 CHASTE_CLASS_EXPORT(ReceptorBipolar)  
146  
147 #endif /* BOIJERETINALNEURALFATES_HPP_ */
```

15.1.21 /src/GomesCellCycleModel.cpp

```

17             rModel.mSeed),
18             ↳ mp_PostMitoticType(rModel.mp_PostMitoticType),
19             ↳ mp_RPh_Type(rModel.mp_RPh_Type), mp_BC_Type(
20             rModel.mp_BC_Type), mp_AC_Type(rModel.mp_AC_Type),
21             ↳ mp_MG_Type(rModel.mp_MG_Type), mp_label_Type(
22             rModel.mp_label_Type)
23 {
24 }
25
26 AbstractCellCycleModel* GomesCellCycleModel::CreateCellCycleModel()
27 {
28     return new GomesCellCycleModel(*this);
29 }
30
31 /**
32 * CELL CYCLE DURATION RANDOM VARIABLE
33 */
34 RandomNumberGenerator* p_random_number_generator =
35     ↳ RandomNumberGenerator::Instance();
36
37 //Gomes cell cycle length determined by lognormal distribution with
38     ↳ default mean 56 hr, std 18.9 hrs.
39 mCellCycleDuration =
40     ↳ exp(p_random_number_generator->NormalRandomDeviate(mNormalMu,
41     ↳ mNormalSigma));
42
43 void GomesCellCycleModel::ResetForDivision()
44 {
45 /**
46 * Mitotic mode rules
47 */
48 RandomNumberGenerator* p_random_number_generator =
49     ↳ RandomNumberGenerator::Instance();
50
51 /**
52 * MITOTIC MODE RANDOM VARIABLE

```

```

52   *****/
53   //initialise mitoticmode random variable, set mitotic mode
54   →  appropriately after comparing to mode probability array
55   double mitoticModeRV = p_random_number_generator→ranf();
56
57   if (mitoticModeRV > mPP && mitoticModeRV ≤ mPP + mPD)
58   {
59     mMitoticMode = 1;
60   }
61   if (mitoticModeRV > mPP + mPD)
62   {
63     mMitoticMode = 2;
64   }
65
66   /**
67   * Write mitotic event to file if appropriate
68   * mDebug: many files: detailed per-lineage info switch; intended for
69   → TestHeInductionCountFixture
70   * mOutput: 1 file: time, seed, cellID, mitotic mode, intended for
71   → TestHeMitoticModeRateFixture
72   * *****/
73
74   if (mDebug)
75   {
76     WriteDebugData(mitoticModeRV);
77   }
78
79   if (mOutput)
80   {
81     WriteModeEventOutput();
82   }
83
84   //set new cell cycle length (will be overwritten with DBL_MAX for DD
85   →  divisions)
86   AbstractSimpleCellCycleModel::ResetForDivision();
87
88   /**
89   * Symmetric postmitotic specification rule
90   * -(asymmetric postmitotic rule specified in
91   → InitialiseDaughterCell());
92   * *****/
93
94   if (mMitoticMode == 2)

```

```

90  {
91      mpCell→SetCellProliferativeType(mp_PostMitoticType);
92      mCellCycleDuration = DBL_MAX;
93      /*****
94      * SPECIFICATION RANDOM VARIABLE
95      *****/
96      double specificationRV = p_random_number_generator→ranf();
97      if (specificationRV ≤ mpMG)
98      {
99          mpCell→AddCellProperty(mp_MG_Type);
100     }
101    if (specificationRV > mpMG && specificationRV ≤ mpMG + mpAC)
102    {
103        mpCell→AddCellProperty(mp_AC_Type);
104    }
105    if (specificationRV > mpMG + mpAC && specificationRV ≤ mpMG +
106        → mpAC + mpBC)
107    {
108        mpCell→AddCellProperty(mp_BC_Type);
109    }
110    if (specificationRV > mpMG + mpAC + mpBC)
111    {
112        mpCell→AddCellProperty(mp_RPh_Type);
113    }
114
115    *****
116    * SEQUENCE SAMPLER
117    *****/
118 //if the sequence sampler has been turned on, check for the label &
119 //→ write mitotic mode to log
120 //50% chance of each daughter cell from a mitosis inheriting the
121 //→ label
122 if (mSequenceSampler)
123 {
124     if (mpCell→HasCellProperty<CellLabel>())
125     {
126         (*LogFile::Instance()) << mMitoticMode;
127         double labelRV = p_random_number_generator→ranf();
128         if (labelRV ≤ .5)
129         {
130             mSeqSamplerLabelSister = true;
131             mpCell→RemoveCellProperty<CellLabel>();

```

```
130         }
131     else
132     {
133         mSeqSamplerLabelSister = false;
134     }
135 }
136 else
137 {
138     //prevents lost-label cells from labelling their progeny
139     mSeqSamplerLabelSister = false;
140 }
141 }
142 }
143
144 void GomesCellCycleModel::InitialiseDaughterCell()
145 {
146     if (mMitoticMode == 0)
147     {
148         //daughter cell's mCellCycleDuration is copied from parent; reset
149         //to new value from gamma PDF here
150         SetCellCycleDuration();
151     }
152
153     *****
154     * PD-type division
155     *****
156
157     if (mMitoticMode == 1)
158     {
159         RandomNumberGenerator* p_random_number_generator =
160             RandomNumberGenerator::Instance();
161         mpCell->SetCellProliferativeType(mp_PostMitoticType);
162         mCellCycleDuration = DBL_MAX;
163
164         *****
165         * SPECIFICATION RULES
166         *****
167         double specificationRV = p_random_number_generator->ranf();
168         if (specificationRV <= mpMG)
169         {
170             mpCell->AddCellProperty(mp_MG_Type);
171         }
172         if (specificationRV > mpMG && specificationRV <= mpMG + mpAC)
173         {
```

```
171         mpCell→AddCellProperty(mp_AC_Type);
172     }
173     if (specificationRV > mpMG + mpAC && specificationRV ≤ mpMG +
174         → mpAC + mpBC)
175     {
176         mpCell→AddCellProperty(mp_BC_Type);
177     }
178     if (specificationRV > mpMG + mpAC + mpBC)
179     {
180         mpCell→AddCellProperty(mp_RPh_Type);
181     }
182
183 if (mMitoticMode = 2)
184 {
185     RandomNumberGenerator* p_random_number_generator =
186         → RandomNumberGenerator::Instance();
187     //remove the fate assigned to the parent cell in
188     → ResetForDivision, then assign the sister fate as usual
189     mpCell→RemoveCellProperty<AbstractCellProperty>();
190     mpCell→SetCellProliferativeType(mp_PostMitoticType);
191
192     /*****
193     * SPECIFICATION RULES
194     *****/
195     double specificationRV = p_random_number_generator→ranf();
196     if (specificationRV ≤ mpMG)
197     {
198         mpCell→AddCellProperty(mp_MG_Type);
199     }
200     if (specificationRV > mpMG && specificationRV ≤ mpMG + mpAC)
201     {
202         mpCell→AddCellProperty(mp_AC_Type);
203     }
204     if (specificationRV > mpMG + mpAC && specificationRV ≤ mpMG +
205         → mpAC + mpBC)
206     {
207         mpCell→AddCellProperty(mp_BC_Type);
208     }
209     if (specificationRV > mpMG + mpAC + mpBC)
210     {
211         mpCell→AddCellProperty(mp_RPh_Type);
212     }
```

```

210     }
211
212     /*****
213     * SEQUENCE SAMPLER
214     *****/
215     if (mSequenceSampler)
216     {
217         if (mSeqSamplerLabelSister)
218         {
219             mpCell->AddCellProperty(mp_label_Type);
220             mSeqSamplerLabelSister = false;
221         }
222         else
223         {
224             mpCell->RemoveCellProperty<CellLabel>();
225         }
226     }
227 }
228
229 void GomesCellCycleModel::SetModelParameters(const double normalMu, const
230                                         → double normalSigma, const double PP,
231                                         → const double PD, const
232                                         → double pBC, const double
233                                         → pAC, const double pMG)
234 {
235     mNormalMu = normalMu;
236     mNormalSigma = normalSigma;
237     mPP = PP;
238     mPD = PD;
239     mpBC = pBC;
240     mpAC = pAC;
241     mpMG = pMG;
242 }
243
244 void
245     → GomesCellCycleModel::SetModelProperties(boost::shared_ptr<AbstractCellProperty>
246     → p_RPh_Type,
247                                         → boost::shared_ptr<AbstractCellProper
248                                         → p_AC_Type,
```



```

276     mSequenceSampler = true;
277     mp_label_Type = label;
278 }
279
280 void
281   ↵ GomesCellCycleModel :: EnableModelDebugOutput(boost :: shared_ptr<ColumnDataWriter>
282   ↵ debugWriter)
283 {
284
285     mDebug = true;
286     mDebugWriter = debugWriter;
287
288     mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");
289     mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
290     mVarIDs.push_back(mDebugWriter->DefineVariable("CycleDuration",
291       ↵ "h"));
292     mVarIDs.push_back(mDebugWriter->DefineVariable("PP", "Percentile"));
293     mVarIDs.push_back(mDebugWriter->DefineVariable("PD", "Percentile"));
294     mVarIDs.push_back(mDebugWriter->DefineVariable("Dieroll",
295       ↵ "Percentile"));
296     mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode",
297       ↵ "Mode"));
298
299     mDebugWriter->EndDefineMode();
300 }
301
302
303 void GomesCellCycleModel :: WriteDebugData(double percentileRoll)
304 {
305     double currentTime = SimulationTime::Instance()->GetTime();
306     CellPtr currentCell = GetCell();
307     double currentCellID = (double) currentCell->GetCellId();
308
309     mDebugWriter->PutVariable(mTimeID, currentTime);
310     mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
311     mDebugWriter->PutVariable(mVarIDs[1], mCellCycleDuration);
312     mDebugWriter->PutVariable(mVarIDs[2], mPP);
313     mDebugWriter->PutVariable(mVarIDs[3], mPD);
314     mDebugWriter->PutVariable(mVarIDs[4], percentileRoll);
315     mDebugWriter->PutVariable(mVarIDs[5], mMitoticMode);
316     mDebugWriter->AdvanceAlongUnlimitedDimension();
317 }
318
319 ****
320 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)

```

```

314  *****/
315
316 double GomesCellCycleModel::GetAverageTransitCellCycleTime()
317 {
318     return (0.0);
319 }
320
321 double GomesCellCycleModel::GetAverageStemCellCycleTime()
322 {
323     return (0.0);
324 }
325
326 void GomesCellCycleModel::OutputCellCycleModelParameters(out_stream&
327   ↪ rParamsFile)
328 {
329     *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
330     ↪ "</CellCycleDuration>\n";
331
332     // Call method on direct parent class
333
334     ↪ AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
335
336 // Serialization for Boost ≥ 1.36
337 #include "SerializationExportWrapperForCpp.hpp"
338 CHASTE_CLASS_EXPORT(GomesCellCycleModel)

```

15.1.22 /src/GomesCellCycleModel.hpp

```

1 #ifndef GOMESCELLCYCLEMODEL_HPP_
2 #define GOMESCELLCYCLEMODEL_HPP_
3
4 #include "AbstractSimpleCellCycleModel.hpp"
5 #include "RandomNumberGenerator.hpp"
6 #include "Cell.hpp"
7 #include "DifferentiatedCellProliferativeType.hpp"
8 #include "GomesRetinalNeuralFates.hpp"
9 #include "SmartPointers.hpp"
10 #include "ColumnDataWriter.hpp"
11 #include "LogFile.hpp"
12 #include "CellLabel.hpp"
13

```

```

14 /*****
15 * GOMES CELL CYCLE MODEL
16 * As described in Gomes et al. 2011 [Gomes2011] doi: 10.1242/dev.059683
17 *
18 * USE: By default, GomesCellCycleModels are constructed with the
19 *      parameter fit reported in [Gomes2011].
20 * Cell cycle length, mitotic mode, and postmitotic fate of cells are
21 *      determined by independent random variables
22 * PP = symmetric proliferative mitotic mode, both progeny remain mitotic
23 * PD = asymmetric proliferative mitotic mode, one progeny exits the cell
24 *      cycle and differentiates
25 * DD = symmetric differentiative mitotic mode, both progeny exit the
26 *      cell cycle and differentiate
27 *
28 * Change default model parameters with SetModelParameters(<params>);
29 * Set AbstractCellProperties for differentiated neural types with
30 *      SetModelProperties();
31 *
32 *
33 * 2 per-model-event output modes:
34 * EnableModeEventOutput() enables mitotic mode event logging-all cells
35 *      will write to the singleton log file
36 * EnableModelDebugOutput() enables more detailed debug output, each seed
37 *      will have its own file written to
38 * by a ColumnDataWriter passed to it from the test
39 * (eg. by the SetupDebugOutput helper function in the project simulator)
40 *
41 * 1 mitotic-event-sequence sampler (only samples one "path" through the
42 *      lineage):
43 * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic
44 *      event type to a string in the singleton log file
45 *
46 *****/
47
48 class GomesCellCycleModel : public AbstractSimpleCellCycleModel
49 {
50     friend class TestSimpleCellCycleModels;
51
52 private:
53
54     /** Needed for serialization.*/
55     friend class boost::serialization::access;
56     /**

```

```
47     * Archive the cell-cycle model and random number generator, never
48     → used directly - boost uses this.
49
50     *
51     * @param archive the archive
52     * @param version the current version of this class
53     */
54
55     template<class Archive>
56     void serialize(Archive & archive, const unsigned int version)
57     {
58         archive &
59         → boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);
60
61         SerializableSingleton<RandomNumberGenerator>* p_wrapper =
62
63             → RandomNumberGenerator::Instance()→GetSerializationWrapper();
64         archive & p_wrapper;
65         archive & mCellCycleDuration;
66     }
67
68     //Private write functions for models
69     void WriteModeEventOutput();
70     void WriteDebugData(double percentile);
71
72 protected:
73     //mode/output variables
74     bool mOutput;
75     double mEventStartTime;
76     bool mSequenceSampler;
77     bool mSeqSamplerLabelSister;
78     //debug writer stuff
79     bool mDebug;
80     int mTimeID;
81     std::vector<int> mVarIDs;
82     boost::shared_ptr<ColumnDataWriter> mDebugWriter;
83     //model parameters and state memory vars
84     double mNormalMu;
85     double mNormalSigma;
86     double mPP;
87     double mPD;
88     double mpBC;
89     double mpAC;
90     double mpMG;
91     unsigned mMitoticMode;
```

```

87     unsigned mSeed;
88     boost::shared_ptr<AbstractCellProperty> mp_PostMitoticType;
89     boost::shared_ptr<AbstractCellProperty> mp_RPh_Type;
90     boost::shared_ptr<AbstractCellProperty> mp_BC_Type;
91     boost::shared_ptr<AbstractCellProperty> mp_AC_Type;
92     boost::shared_ptr<AbstractCellProperty> mp_MG_Type;
93     boost::shared_ptr<AbstractCellProperty> mp_label_Type;
94
95     /**
96      * Protected copy-constructor for use by CreateCellCycleModel().
97      *
98      * The only way for external code to create a copy of a cell cycle
99      * model
100     * is by calling that method, to ensure that a model of the correct
101     * subclass is created.
102     * This copy-constructor helps subclasses to ensure that all member
103     * variables are correctly copied when this happens.
104     *
105     * This method is called by child classes to set member variables for
106     * a daughter cell upon cell division.
107     * Note that the parent cell cycle model will have had
108     * ResetForDivision() called just before CreateCellCycleModel() is
109     * called,
110     * so performing an exact copy of the parent is suitable behaviour.
111     * Any daughter-cell-specific initialisation
112     * can be done in InitialiseDaughterCell().
113     */
114     GomesCellCycleModel(const GomesCellCycleModel& rModel);
115
116 public:
117
118     /**
119      * Constructor - just a default, mBirthTime is set in the
120      * AbstractCellCycleModel class.
121      */
122     GomesCellCycleModel();
123
124     /**
125      * SetCellCycleDuration() method to set length of cell cycle
126      * (lognormal distribution as specified in [Gomes2011])
127      */
128
129
130

```

```

121 void SetCellCycleDuration();
122
123 /**
124 * Overridden builder method to create new copies of
125 * this cell-cycle model.
126 *
127 * @return new cell-cycle model
128 */
129 AbstractCellCycleModel* CreateCellCycleModel();
130
131 /**
132 * Overridden ResetForDivision() method.
133 * Contains general mitotic mode logic
134 */
135 void ResetForDivision();
136
137 /** Overridden InitialiseDaughterCell() method. Used to implement
138 * asymmetric mitotic mode*/
139 void InitialiseDaughterCell();
140
141 /* Model setup functions
142 * Set lognormal cell cycle curve properties with *mean and std of
143 * corresponding NORMAL curve*
144 * Model requires valid AbstractCellProperties to assign postmitotic
145 * fate;
146 * Defaults are found in GomesRetinalNeuralFates.hpp
147 * (RodPhotoreceptor, AmacrineCell, BipolarCell, MullerGlia)
148 */
149
150 void SetModelParameters(const double normalMu = 3.9716, const double
151 * normalSigma = .32839, const double PP = .055,
152 * const double PD = .221, const double pBC =
153 * .128, const double pAC = .106, const
154 * double pMG =
155 * .028);
156
157 void SetModelProperties(boost::shared_ptr<AbstractCellProperty>
158 * p_RPh_Type,
159 * boost::shared_ptr<AbstractCellProperty>
160 * p_AC_Type,
161 * boost::shared_ptr<AbstractCellProperty>
162 * p_BC_Type,
163 * boost::shared_ptr<AbstractCellProperty>
164 * p_MG_Type);

```

```

154
155 //This should normally be a DifferentiatedCellProliferativeType
156 void SetPostMitoticType(boost::shared_ptr<AbstractCellProperty>
157   → p_PostMitoticType);
158
159 //Functions to enable per-cell mitotic mode logging for mode rate &
160   → sequence sampling fixtures
161 //Uses singleton logfile
162 void EnableModeEventOutput(double eventStart, unsigned seed);
163 void EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty>
164   → label);
165
166 //More detailed debug output. Needs a ColumnDataWriter passed to it
167 //Only declare ColumnDataWriter directory, filename, etc; do not set
168   → up otherwise
169 void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
170   → debugWriter);
171
172 //Not used, but must be overwritten lest GomesCellCycleModels be
173   → abstract
174 double GetAverageTransitCellCycleTime();
175 double GetAverageStemCellCycleTime();
176
177 /**
178 * Overridden OutputCellCycleModelParameters() method.
179 *
180 * @param rParamsFile the file stream to which the parameters are
181   → output
182 */
183 virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
184 };
185
186 #include "SerializationExportWrapper.hpp"
187 // Declare identifier for the serializer
188 CHASTE_CLASS_EXPORT(GomesCellCycleModel)
189
190
191 #endif /*GOMESCELLCYCLEMODEL_HPP_*/

```

15.1.23 /src/GomesRetinalNeuralFates.cpp

```

1 #include "GomesRetinalNeuralFates.hpp"
2

```

```
3 RodPhotoreceptor::RodPhotoreceptor(unsigned colour)
4     : AbstractCellProperty(),
5         mColour(colour)
6 {
7 }
8
9 RodPhotoreceptor::~RodPhotoreceptor()
10 {
11 }
12
13 unsigned RodPhotoreceptor::GetColour() const
14 {
15     return mColour;
16 }
17
18 AmacrineCell::AmacrineCell(unsigned colour)
19     : AbstractCellProperty(),
20         mColour(colour)
21 {
22 }
23
24 AmacrineCell::~AmacrineCell()
25 {
26 }
27
28 unsigned AmacrineCell::GetColour() const
29 {
30     return mColour;
31 }
32
33 BipolarCell::BipolarCell(unsigned colour)
34     : AbstractCellProperty(),
35         mColour(colour)
36 {
37 }
38
39 BipolarCell::~BipolarCell()
40 {
41 }
42
43 unsigned BipolarCell::GetColour() const
44 {
45     return mColour;
```

```

46 }
47
48 MullerGlia::MullerGlia(unsigned colour)
49     : AbstractCellProperty(),
50     mColour(colour)
51 {
52 }
53
54 MullerGlia::~MullerGlia()
55 {
56 }
57
58 unsigned MullerGlia::GetColour() const
59 {
60     return mColour;
61 }
62
63 #include "SerializationExportWrapperForCpp.hpp"
64 // Declare identifier for the serializer
65 CHASTE_CLASS_EXPORT(RodPhotoreceptor)
66 CHASTE_CLASS_EXPORT(AmacrineCell)
67 CHASTE_CLASS_EXPORT(BipolarCell)
68 CHASTE_CLASS_EXPORT(MullerGlia)

```

15.1.24 /src/GomesRetinalNeuralFates.hpp

```

1 #ifndef GOMESRETINALNEURALFATES_HPP_
2 #define GOMESRETINALNEURALFATES_HPP_
3
4 #include <boost/shared_ptr.hpp>
5 #include "AbstractCellProperty.hpp"
6 #include "ChasteSerialization.hpp"
7 #include <boost/serialization/base_object.hpp>
8
9 class RodPhotoreceptor : public AbstractCellProperty
10 {
11 private:
12
13     /**
14      * Colour for use by visualizer.
15      */
16     unsigned mColour;

```

```
17
18     /** Needed for serialization. */
19     friend class boost::serialization::access;
20
21     /**
22      * Archive the member variables.
23      *
24      * @param archive the archive
25      * @param version the current version of this class
26      */
27     template<class Archive>
28     void serialize(Archive & archive, const unsigned int version)
29     {
30         archive &
31             boost::serialization::base_object<AbstractCellProperty>(*this);
32         archive & mColour;
33     }
34
35 public:
36
37     /**
38      * Constructor.
39      *
40      * @param colour what colour cells with this property should be in
41      * the visualizer (defaults to 6)
42      */
43     RodPhotoreceptor(unsigned colour=3);
44
45     virtual ~RodPhotoreceptor();
46
47     /**
48      * @return #mColour.
49      */
50     unsigned GetColour() const;
51 };
52
53 class AmacrineCell : public AbstractCellProperty
54 {
55 private:
56
57     /**
```

```
58     * Colour for use by visualizer.  
59     */  
60     unsigned mColour;  
61  
62     /** Needed for serialization. */  
63     friend class boost::serialization::access;  
64     /**  
65      * Archive the member variables.  
66      *  
67      * @param archive the archive  
68      * @param version the current version of this class  
69      */  
70     template<class Archive>  
71     void serialize(Archive & archive, const unsigned int version)  
72     {  
73         archive &  
74             boost::serialization::base_object<AbstractCellProperty>(*this);  
75         archive & mColour;  
76     }  
77  
78 public:  
79  
80     /**  
81      * Constructor.  
82      *  
83      * @param colour what colour cells with this property should be in  
84      * the visualizer (defaults to 6)  
85      */  
86     AmacrineCell(unsigned colour=4);  
87  
88     /**  
89      * Destructor.  
90      */  
91     virtual ~AmaCrineCell();  
92  
93     /**  
94      * @return #mColour.  
95      */  
96     unsigned GetColour() const;  
97 };  
98  
99 class BipolarCell : public AbstractCellProperty  
100 {
```

```
99 private:
100
101     /**
102      * Colour for use by visualizer.
103      */
104     unsigned mColour;
105
106     /** Needed for serialization.*/
107     friend class boost::serialization::access;
108     /**
109      * Archive the member variables.
110      *
111      * @param archive the archive
112      * @param version the current version of this class
113      */
114     template<class Archive>
115     void serialize(Archive & archive, const unsigned int version)
116     {
117         archive &
118             → boost::serialization::base_object<AbstractCellProperty>(*this);
119         archive & mColour;
120     }
121
122
123 public:
124
125     /**
126      * Constructor.
127      *
128      * @param colour what colour cells with this property should be in
129      * the visualizer (defaults to 6)
130      */
131     BipolarCell(unsigned colour=5);
132
133     /**
134      * Destructor.
135      */
136     virtual ~BipolarCell();
137
138     /**
139      * @return #mColour.
140      */
141     unsigned GetColour() const;
142 }
```

```
140
141 class MullerGlia : public AbstractCellProperty
142 {
143 private:
144
145     /**
146      * Colour for use by visualizer.
147      */
148     unsigned mColour;
149
150     /** Needed for serialization.*/
151     friend class boost::serialization::access;
152     /**
153      * Archive the member variables.
154      *
155      * @param archive the archive
156      * @param version the current version of this class
157      */
158     template<class Archive>
159     void serialize(Archive & archive, const unsigned int version)
160     {
161         archive &
162             boost::serialization::base_object<AbstractCellProperty>(*this);
163         archive & mColour;
164     }
165
166 public:
167
168     /**
169      * Constructor.
170      *
171      * @param colour what colour cells with this property should be in
172      * the visualizer (defaults to 6)
173      */
174     MullerGlia(unsigned colour=6);
175
176     /**
177      * Destructor.
178      */
179     virtual ~MullerGlia();
180
181     /**
182      * @return #mColour.
```

```

181     */
182     unsigned GetColour() const;
183 };
184 #include "SerializationExportWrapper.hpp"
185 // Declare identifier for the serializer
186 CHASTE_CLASS_EXPORT(RodPhotoreceptor)
187 CHASTE_CLASS_EXPORT(AmacrineCell)
188 CHASTE_CLASS_EXPORT(BipolarCell)
189 CHASTE_CLASS_EXPORT(MullerGlia)
190
191 #endif /* GOMESRETINALNEURALFATES_HPP_ */

```

15.1.25 /src/HeAth5Mo.cpp

```

1 #include "../..../..../projects/ISP/src/HeAth5Mo.hpp"
2
3 Ath5Mo::Ath5Mo(unsigned colour)
4     : AbstractCellProperty(),
5      mColour(colour)
6 {
7 }
8
9 Ath5Mo::~Ath5Mo()
10 {
11 }
12
13 unsigned Ath5Mo::GetColour() const
14 {
15     return mColour;
16 }
17
18 #include "SerializationExportWrapperForCpp.hpp"
19 // Declare identifier for the serializer
20 CHASTE_CLASS_EXPORT(Ath5Mo)

```

15.1.26 /src/HeAth5Mo.hpp

```

1 #ifndef HEATH5MO_HPP_
2 #define HEATH5MO_HPP_
3
4 #include <boost/shared_ptr.hpp>

```

```
5 #include "AbstractCellProperty.hpp"
6 #include "ChasteSerialization.hpp"
7 #include <boost/serialization/base_object.hpp>
8
9 class Ath5Mo : public AbstractCellProperty
10 {
11 private:
12
13     /**
14      * Colour for use by visualizer.
15      */
16     unsigned mColour;
17
18     /** Needed for serialization.*/
19     friend class boost::serialization::access;
20
21     /**
22      * Archive the member variables.
23      *
24      * @param archive the archive
25      * @param version the current version of this class
26      */
27     template<class Archive>
28     void serialize(Archive & archive, const unsigned int version)
29     {
30         archive &
31             boost::serialization::base_object<AbstractCellProperty>(*this);
32         archive & mColour;
33     }
34
35 public:
36
37     /**
38      * Constructor.
39      *
40      * @param colour what colour cells with this property should be in
41      * the visualizer (defaults to 6)
42      */
43     Ath5Mo(unsigned colour=3);
44
45     /**
46      * Destructor.
47      */
48     virtual ~Ath5Mo();
```

```

46
47     /**
48      * @return #mColour.
49      */
50     unsigned GetColour() const;
51 }
52
53
54 #include "SerializationExportWrapper.hpp"
55 // Declare identifier for the serializer
56 CHASTE_CLASS_EXPORT(Ath5Mo)
57 #endif /* HEATH5MO_HPP */

```

15.1.27 /src/HeCellCycleModel.cpp

```

1 #include "HeCellCycleModel.hpp"
2
3 HeCellCycleModel::HeCellCycleModel() :
4     AbstractSimpleCellCycleModel(), mKillSpecified(false),
5     mDeterministic(false), mOutput(false), mEventStartTime(
6         24.0), mSequenceSampler(false),
7     mSeqSamplerLabelSister(false), mDebug(false),
8     mTimeID(), mVarIDs(), mDebugWriter(), mTiLOffset(
9         0.0), mGammaShift(4.0), mGammaShape(2.0),
10    mGammaScale(1.0), mSisterShiftWidth(1),
11    mMitoticModePhase2(
12        8.0), mMitoticModePhase3(15.0), mPhaseShiftWidth(2.0),
13    mPhase1PP(1.0), mPhase1PD(0.0), mPhase2PP(0.2),
14    mPhase2PD(
15        0.4), mPhase3PP(0.2), mPhase3PD(0.0), mMitoticMode(0),
16    mSeed(0), mTimeDependentCycleDuration(false),
17    mPeakRateTime(), mIncreasingRateSlope(),
18    mDecreasingRateSlope(), mBaseGammaScale()
19 {
20     mReadyToDivide = true; //He model begins with a first division
21 }
22
23 HeCellCycleModel::HeCellCycleModel(const HeCellCycleModel& rModel) :
24     AbstractSimpleCellCycleModel(rModel),
25     mKillSpecified(rModel.mKillSpecified), mDeterministic(

```

```

15         rModel.mDeterministic), mOutput(rModel.mOutput),
16         ↳ mEventStartTime(rModel.mEventStartTime),
17         ↳ mSequenceSampler(
18             rModel.mSequenceSampler),
19             ↳ mSeqSamplerLabelSister(rModel.mSeqSamplerLabelSister),
20             ↳ mDebug(rModel.mDebug), mTimeID(
21                 rModel.mTimeID), mVarIDs(rModel.mVarIDs),
22                 ↳ mDebugWriter(rModel.mDebugWriter), mTiLOffset(
23                     rModel.mTiLOffset), mGammaShift(rModel.mGammaShift),
24                     ↳ mGammaShape(rModel.mGammaShape), mGammaScale(
25                         rModel.mGammaScale),
26                         ↳ mSisterShiftWidth(rModel.mSisterShiftWidth),
27                         ↳ mMitoticModePhase2(
28                             rModel.mMitoticModePhase2),
29                             ↳ mMitoticModePhase3(rModel.mMitoticModePhase3),
30                             ↳ mPhaseShiftWidth(
31                                 rModel.mPhaseShiftWidth), mPhase1PP(rModel.mPhase1PP),
32                                 ↳ mPhase1PD(rModel.mPhase1PD), mPhase2PP(
33                                     rModel.mPhase2PP), mPhase2PD(rModel.mPhase2PD),
34                                     ↳ mPhase3PP(rModel.mPhase3PP), mPhase3PD(
35                                         rModel.mPhase3PD), mMitoticMode(rModel.mMitoticMode),
36                                         ↳ mSeed(rModel.mSeed), mTimeDependentCycleDuration(
37                                             rModel.mTimeDependentCycleDuration),
38                                             ↳ mPeakRateTime(rModel.mPeakRateTime),
39                                             ↳ mIncreasingRateSlope(
40                                                 rModel.mIncreasingRateSlope),
41                                                 ↳ mDecreasingRateSlope(rModel.mDecreasingRateSlope),
42                                                 ↳ mBaseGammaScale(
43                                                     rModel.mBaseGammaScale)
44     {
45 }
46
47 AbstractCellCycleModel* HeCellCycleModel::CreateCellCycleModel()
48 {
49     return new HeCellCycleModel(*this);
50 }
51
52 void HeCellCycleModel::SetCellCycleDuration()
53 {
54     RandomNumberGenerator* p_random_number_generator =
55         ↳ RandomNumberGenerator::Instance();
56
57 /*****
58 *****/

```

```

40   * CELL CYCLE DURATION RANDOM VARIABLE
41   *****/
42
43   if (!mTimeDependentCycleDuration) //Normal operation, cell cycle
44     ↳ length stays constant
45   {
46     //He cell cycle length determined by shifted gamma distribution
47     ↳ reflecting 4 hr refractory period followed by gamma pdf
48     mCellCycleDuration = mGammaShift +
49       ↳ p_random_number_generator→GammaRandomDeviate(mGammaShape,
50         ↳ mGammaScale);
51
52   }
53
54   ****
55   * Variable cycle length
56   * Give -ve mIncreasingRateSlope and +ve mDecreasingRateSlope,
57   * cell cycle length linearly declines (increasing rate), then
58   ↳ increases, switching at mPeakRateTime
59   ****/
60
61   else
62   {
63     double currTime = SimulationTime::Instance()→GetTime();
64     if (currTime ≤ mPeakRateTime)
65     {
66       mGammaScale = std::max((mBaseGammaScale - currTime *
67         ↳ mIncreasingRateSlope), .000000000001);
68     }
69     if (currTime > mPeakRateTime)
70     {
71       mGammaScale = std::max(
72         ((mBaseGammaScale - mPeakRateTime *
73           ↳ mIncreasingRateSlope)
74           + (mBaseGammaScale + (currTime -
75             ↳ mPeakRateTime) * mDecreasingRateSlope)),
76           .000000000001);
77     }
78     mCellCycleDuration = mGammaShift +
79       ↳ p_random_number_generator→GammaRandomDeviate(mGammaShape,
80         ↳ mGammaScale);
81   }
82
83 }
```

```

73 void HeCellCycleModel::ResetForDivision()
74 {
75     /**************************************************************************
76     * TIME IN LINEAGE DEPENDENT MITOTIC MODE PHASE RULES
77     * ****
78     RandomNumberGenerator* p_random_number_generator =
79         → RandomNumberGenerator::Instance();
80
81     double currentTiL = SimulationTime::Instance()→GetTime() +
82         → mTiLOffset;
83
84     /*Rule logic defaults to phase 1 behaviour, checks for currentTiL >
85      → phaseBoundaries and changes
86      currentPhase and subsequently mMitoticMode as appropriate*/
87     unsigned currentPhase = 1;
88     mMitoticMode = 0;
89
90     //Check time in lineage and determine current mitotic mode phase
91     ****
92     * Phase boundary & deterministic mitotic mode rules
93     ****
94     if (currentTiL > mMitoticModePhase2 && currentTiL <
95         → mMitoticModePhase3)
96     {
97         //if current TiL is > phase 2 boundary time, set the currentPhase
98         → appropriately
99         currentPhase = 2;
100
101         //if deterministic mode is enabled, PD divisions are guaranteed
102         → unless this is an Ath5 morphant
103         if (mDeterministic)
104         {
105             mMitoticMode = 1; //0=PP;1=PD;2=DD
106             if (mpCell→HasCellProperty<Ath5Mo>()) //Ath5 morphants
107                 → undergo PP rather than PD divisions in 80% of cases
108             {
109                 double ath5RV = p_random_number_generator→ranf();
110                 if (ath5RV ≤ .8)
111                 {
112                     mMitoticMode = 0;
113                 }
114             }
115         }
116     }
117 }
```

```

109 }
110
111 if (currentTiL > mMitoticModePhase3)
112 {
113     //if current TiL is > phase 3 boundary time, set the currentPhase
114     //appropriately
115     currentPhase = 3;
116     if (mDeterministic)
117     {
118         //if deterministic mode is enabled, DD divisions are
119         //guaranteed
120         mMitoticMode = 2;
121     }
122 }
123
124 /**
125 * MITOTIC MODE RANDOM VARIABLE
126 *****/
127 //initialise mitoticmode random variable, set mitotic mode
128 //appropriately after comparing to mode probability matrix
129 double mitoticModeRV = p_random_number_generator->ranf(); //0-1
130 //evenly distributed RV
131
132 if (!mDeterministic)
133 {
134     //construct 3x2 matrix of mode probabilities arranged by phase
135     double modeProbabilityMatrix[3][2] = { { mPhase1PP, mPhase1PD },
136                                         { mPhase2PP, mPhase2PD }, { mPhase3PP,
137
138
139     //if the RV is > currentPhasePP && <= currentPhasePD, change
140     //mMitoticMode from PP to PD
141     if (mitoticModeRV > modeProbabilityMatrix[currentPhase - 1][0]
142         && mitoticModeRV
143             <= modeProbabilityMatrix[currentPhase - 1][0] +
144                 modeProbabilityMatrix[currentPhase - 1][1])
145     {
146         mMitoticMode = 1;
147         if (mpCell->HasCellProperty<Ath5Mo>()) //Ath5 morphants
148             //undergo PP rather than PD divisions in 80% of cases

```

```

141         {
142             double ath5RV = p_random_number_generator->ranf();
143             if (ath5RV <=.8)
144             {
145                 mMitoticMode = 0;
146             }
147         }
148     }
149     //if the RV is > currentPhasePP + currentPhasePD, change
150     //→ mMitoticMode from PP to DD
151     if (mitoticModeRV > modeProbabilityMatrix[currentPhase - 1][0] +
152         modeProbabilityMatrix[currentPhase - 1][1])
153     {
154         mMitoticMode = 2;
155     }
156 /**
157 * Write mitotic event to relevant files
158 */
159 if (mDebug)
160 {
161     WriteDebugData(currentTiL, currentPhase, mitoticModeRV);
162 }
163
164 if (mOutput)
165 {
166     WriteModeEventOutput();
167 }
168
169 //set new cell cycle length (will be overwritten with DBL_MAX for DD
170 //→ divisions)
171 AbstractSimpleCellCycleModel::ResetForDivision();
172 /**
173 * Symmetric postmitotic specification rule
174 * -(asymmetric postmitotic rule specified in
175 //→ InitialiseDaughterCell());
176 */
177 if (mMitoticMode == 2)
178 {
179     boost::shared_ptr<AbstractCellProperty> p_PostMitoticType =

```

```

179             → mpCell→rGetCellPropertyCollection().GetCellPropertyRegistry()→G
180             mpCell→SetCellProliferativeType(p_PostMitoticType);
181             mCellCycleDuration = DBL_MAX;
182
183         if(mKillSpecified)
184         {
185             mpCell→Kill();
186         }
187     }
188
189     /*****
190      * SEQUENCE SAMPLER
191      *****/
192     //if the sequence sampler has been turned on, check for the label &
193     → write mitotic mode to log
194     //50% chance of each daughter cell from a mitosis inheriting the
195     → label
196     if (mSequenceSampler)
197     {
198         if (mpCell→HasCellProperty<CellLabel>())
199         {
200             (*LogFile::Instance()) << mMitoticMode;
201             double labelRV = p_random_number_generator→ranf();
202             if (labelRV ≤ .5)
203             {
204                 mSeqSamplerLabelSister = true;
205                 mpCell→RemoveCellProperty<CellLabel>();
206             }
207             else
208             {
209                 mSeqSamplerLabelSister = false;
210             }
211         }
212         else
213         {
214             //prevents lost-label cells from labelling their progeny
215             mSeqSamplerLabelSister = false;
216         }
217     }
218 void HeCellCycleModel::Initialise()

```

```

219 {
220     boost::shared_ptr<AbstractCellProperty> p_Transit =
221
222         ↪ mpCell→rGetCellPropertyCollection().GetCellPropertyRegistry()→Get<T>
223     mpCell→SetCellProliferativeType(p_Transit);
224
225     if (mTiLOffset == 0) //the "regular" case, set cycle duration
226         ↪ normally
227     {
228         SetCellCycleDuration();
229     }
230
231     if (mTiLOffset < 0) //these cells are offspring of Wan stem cells
232     {
233         mReadyToDivide = false;
234         SetCellCycleDuration();
235     }
236
237     if (mTiLOffset > 0.0) //if the TiL is > 0, the first division has
238         ↪ already occurred
239     {
240         mReadyToDivide = false;
241
242         RandomNumberGenerator* p_random_number_generator =
243             ↪ RandomNumberGenerator::Instance();
244
245         /**
246             * This calculation "runs time forward" by subtracting
247             ↪ appropriately generated cell lengths from TiLOffset
248                 * Ultimately c is subtracted from a final cell length
249                 ↪ calculation to give the appropriate reduced cycle length
250             */
251
252         double c = mTiLOffset;
253         while (c > 0)
254         {
255             c = c - (mGammaShift +
256                 ↪ p_random_number_generator→GammaRandomDeviate(mGammaShape,
257                 ↪ mGammaScale));
258         }
259
260
261

```

```

252     mCellCycleDuration = (mGammaShift +
253         ↳ p_random_number_generator→GammaRandomDeviate(mGammaShape,
254             ↳ mGammaScale))
255         + c;
256     }
257
258 void HeCellCycleModel::InitialiseDaughterCell()
259 {
260     RandomNumberGenerator* p_random_number_generator =
261         ↳ RandomNumberGenerator::Instance();
262
263     /*****
264      * PD-type division & shifted sister cycle length & boundary
265      * adjustments
266      *****/
267
268     if (mMitoticMode == 1) //RPC becomes specified retinal neuron in
269         ↳ asymmetric PD mitosis
270     {
271         boost::shared_ptr<AbstractCellProperty> p_PostMitoticType =
272
273             ↳ mpCell→rGetCellPropertyCollection().GetCellPropertyRegistry()→G
274         mpCell→SetCellProliferativeType(p_PostMitoticType);
275         mCellCycleDuration = DBL_MAX;
276
277         if(mKillSpecified)
278         {
279             mpCell→Kill();
280         }
281
282         //daughter cell's mCellCycleDuration is copied from parent; modified
283         ↳ by a normally distributed shift if it remains proliferative
284     if (mMitoticMode == 0)
285     {
286         double sisterShift =
287             ↳ p_random_number_generator→NormalRandomDeviate(0,
288                 ↳ mSisterShiftWidth); //random variable mean 0 SD 1 by default
289         mCellCycleDuration = std::max(mGammaShift, mCellCycleDuration +
290             ↳ sisterShift); // sister shift respects 4 hour refractory
291             ↳ period

```

```

284 }
285
286 //deterministic model phase boundary division shift for daughter
287 // cells
288 if (mDeterministic)
289 {
290     //shift phase boundaries to reflect error in "timer" after
291     // division
292     double phaseShift =
293         p_random_number_generator->NormalRandomDeviate(0,
294         mPhaseShiftWidth);
295     mMitoticModePhase2 = mMitoticModePhase2 + phaseShift;
296     mMitoticModePhase3 = mMitoticModePhase3 + phaseShift;
297 }
298
299 /**
300 * SEQUENCE SAMPLER
301 *****/
302 if (mSequenceSampler)
303 {
304     if (mSeqSamplerLabelSister)
305     {
306         boost::shared_ptr<AbstractCellProperty> p_label_type =
307
308             mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry(
309             mpCell->AddCellProperty(p_label_type);
310             mSeqSamplerLabelSister = false;
311     }
312     else
313     {
314         mpCell->RemoveCellProperty<CellLabel>();
315     }
316 }
317
318 if (mMitoticMode == 2 && mKillSpecified) mpCell->Kill();
319 }
320
321 void HeCellCycleModel::SetModelParameters(double t1Offset, double
322     mitoticModePhase2, double mitoticModePhase3,
323
324     double phase1PP, double
325     phase1PD, double phase2PP,
326     double phase2PD,
327
328     double phase3PP, double
329     phase3PD, double phase4PP,
330     double phase4PD,
331
332     double phase5PP, double
333     phase5PD, double phase6PP,
334     double phase6PD,
335
336     double phase7PP, double
337     phase7PD, double phase8PP,
338     double phase8PD,
339
340     double phase9PP, double
341     phase9PD, double phase10PP,
342     double phase10PD,
343
344     double phase11PP, double
345     phase11PD, double phase12PP,
346     double phase12PD,
347
348     double phase13PP, double
349     phase13PD, double phase14PP,
350     double phase14PD,
351
352     double phase15PP, double
353     phase15PD, double phase16PP,
354     double phase16PD,
355
356     double phase17PP, double
357     phase17PD, double phase18PP,
358     double phase18PD,
359
360     double phase19PP, double
361     phase19PD, double phase20PP,
362     double phase20PD,
363
364     double phase21PP, double
365     phase21PD, double phase22PP,
366     double phase22PD,
367
368     double phase23PP, double
369     phase23PD, double phase24PP,
370     double phase24PD,
371
372     double phase25PP, double
373     phase25PD, double phase26PP,
374     double phase26PD,
375
376     double phase27PP, double
377     phase27PD, double phase28PP,
378     double phase28PD,
379
380     double phase29PP, double
381     phase29PD, double phase30PP,
382     double phase30PD,
383
384     double phase31PP, double
385     phase31PD, double phase32PP,
386     double phase32PD,
387
388     double phase33PP, double
389     phase33PD, double phase34PP,
390     double phase34PD,
391
392     double phase35PP, double
393     phase35PD, double phase36PP,
394     double phase36PD,
395
396     double phase37PP, double
397     phase37PD, double phase38PP,
398     double phase38PD,
399
400     double phase39PP, double
401     phase39PD, double phase40PP,
402     double phase40PD,
403
404     double phase41PP, double
405     phase41PD, double phase42PP,
406     double phase42PD,
407
408     double phase43PP, double
409     phase43PD, double phase44PP,
410     double phase44PD,
411
412     double phase45PP, double
413     phase45PD, double phase46PP,
414     double phase46PD,
415
416     double phase47PP, double
417     phase47PD, double phase48PP,
418     double phase48PD,
419
420     double phase49PP, double
421     phase49PD, double phase50PP,
422     double phase50PD,
423
424     double phase51PP, double
425     phase51PD, double phase52PP,
426     double phase52PD,
427
428     double phase53PP, double
429     phase53PD, double phase54PP,
430     double phase54PD,
431
432     double phase55PP, double
433     phase55PD, double phase56PP,
434     double phase56PD,
435
436     double phase57PP, double
437     phase57PD, double phase58PP,
438     double phase58PD,
439
440     double phase59PP, double
441     phase59PD, double phase60PP,
442     double phase60PD,
443
444     double phase61PP, double
445     phase61PD, double phase62PP,
446     double phase62PD,
447
448     double phase63PP, double
449     phase63PD, double phase64PP,
450     double phase64PD,
451
452     double phase65PP, double
453     phase65PD, double phase66PP,
454     double phase66PD,
455
456     double phase67PP, double
457     phase67PD, double phase68PP,
458     double phase68PD,
459
460     double phase69PP, double
461     phase69PD, double phase70PP,
462     double phase70PD,
463
464     double phase71PP, double
465     phase71PD, double phase72PP,
466     double phase72PD,
467
468     double phase73PP, double
469     phase73PD, double phase74PP,
470     double phase74PD,
471
472     double phase75PP, double
473     phase75PD, double phase76PP,
474     double phase76PD,
475
476     double phase77PP, double
477     phase77PD, double phase78PP,
478     double phase78PD,
479
480     double phase79PP, double
481     phase79PD, double phase80PP,
482     double phase80PD,
483
484     double phase81PP, double
485     phase81PD, double phase82PP,
486     double phase82PD,
487
488     double phase83PP, double
489     phase83PD, double phase84PP,
490     double phase84PD,
491
492     double phase85PP, double
493     phase85PD, double phase86PP,
494     double phase86PD,
495
496     double phase87PP, double
497     phase87PD, double phase88PP,
498     double phase88PD,
499
500     double phase89PP, double
501     phase89PD, double phase90PP,
502     double phase90PD,
503
504     double phase91PP, double
505     phase91PD, double phase92PP,
506     double phase92PD,
507
508     double phase93PP, double
509     phase93PD, double phase94PP,
510     double phase94PD,
511
512     double phase95PP, double
513     phase95PD, double phase96PP,
514     double phase96PD,
515
516     double phase97PP, double
517     phase97PD, double phase98PP,
518     double phase98PD,
519
520     double phase99PP, double
521     phase99PD, double phase100PP,
522     double phase100PD,
523
524     double phase101PP, double
525     phase101PD, double phase102PP,
526     double phase102PD,
527
528     double phase103PP, double
529     phase103PD, double phase104PP,
530     double phase104PD,
531
532     double phase105PP, double
533     phase105PD, double phase106PP,
534     double phase106PD,
535
536     double phase107PP, double
537     phase107PD, double phase108PP,
538     double phase108PD,
539
540     double phase109PP, double
541     phase109PD, double phase110PP,
542     double phase110PD,
543
544     double phase111PP, double
545     phase111PD, double phase112PP,
546     double phase112PD,
547
548     double phase113PP, double
549     phase113PD, double phase114PP,
550     double phase114PD,
551
552     double phase115PP, double
553     phase115PD, double phase116PP,
554     double phase116PD,
555
556     double phase117PP, double
557     phase117PD, double phase118PP,
558     double phase118PD,
559
560     double phase119PP, double
561     phase119PD, double phase120PP,
562     double phase120PD,
563
564     double phase121PP, double
565     phase121PD, double phase122PP,
566     double phase122PD,
567
568     double phase123PP, double
569     phase123PD, double phase124PP,
570     double phase124PD,
571
572     double phase125PP, double
573     phase125PD, double phase126PP,
574     double phase126PD,
575
576     double phase127PP, double
577     phase127PD, double phase128PP,
578     double phase128PD,
579
580     double phase129PP, double
581     phase129PD, double phase130PP,
582     double phase130PD,
583
584     double phase131PP, double
585     phase131PD, double phase132PP,
586     double phase132PD,
587
588     double phase133PP, double
589     phase133PD, double phase134PP,
590     double phase134PD,
591
592     double phase135PP, double
593     phase135PD, double phase136PP,
594     double phase136PD,
595
596     double phase137PP, double
597     phase137PD, double phase138PP,
598     double phase138PD,
599
599 }
```

```

318                               double phase3PP, double
319                               ↵ phase3PD, double
320                               ↵ gammaShift, double
321                               ↵ gammaShape,
322                               double gammaScale, double
323                               ↵ sisterShift)
324 {
325     mTiLOffset = tiLOffset;
326     mMitoticModePhase2 = mitoticModePhase2;
327     mMitoticModePhase3 = mitoticModePhase3;
328     mPhase1PP = phase1PP;
329     mPhase1PD = phase1PD;
330     mPhase2PP = phase2PP;
331     mPhase2PD = phase2PD;
332     mPhase3PP = phase3PP;
333     mPhase3PD = phase3PD;
334     mGammaShift = gammaShift;
335     mGammaShape = gammaShape;
336     mGammaScale = gammaScale;
337     mSisterShiftWidth = sisterShift;
338 }
339
340 void HeCellCycleModel::SetDeterministicMode(double tiLOffset, double
341     ↵ mitoticModePhase2, double mitoticModePhase3,
342                               double phaseShiftWidth,
343                               ↵ double gammaShift, double
344                               ↵ gammaShape,
345                               double gammaScale, double
346                               ↵ sisterShift)
347 {
348     mDeterministic = true;
349     mTiLOffset = tiLOffset;
350     mMitoticModePhase2 = mitoticModePhase2;
351     mMitoticModePhase3 = mitoticModePhase3;
352     mPhaseShiftWidth = phaseShiftWidth;
353     mGammaShift = gammaShift;
354     mGammaShape = gammaShape;
355     mGammaScale = gammaScale;
356     mSisterShiftWidth = sisterShift;
357 }
358
359 void HeCellCycleModel::SetTimeDependentCycleDuration(double peakRateTime,
360     ↵ double increasingSlope,

```

```

352                               double
353                               ↵ decreasingSlope)
354
355     mTimeDependentCycleDuration = true;
356     mPeakRateTime = peakRateTime;
357     mIncreasingRateSlope = increasingSlope;
358     mDecreasingRateSlope = decreasingSlope;
359     mBaseGammaScale = mGammaScale;
360 }
361
362 void HeCellCycleModel::EnableKillSpecified()
363 {
364     mKillSpecified = true;
365 }
366
367 void HeCellCycleModel::EnableModeEventOutput(double eventStart, unsigned
368     ↵ seed)
369 {
370     mOutput = true;
371     mEventStartTime = eventStart;
372     mSeed = seed;
373 }
374
375 void HeCellCycleModel::WriteModeEventOutput()
376 {
377     double currentTime = SimulationTime::Instance()->GetTime() +
378         ↵ mEventStartTime;
379     CellPtr currentCell = GetCell();
380     double currentCellID = (double) currentCell->GetCellId();
381     (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" <<
382         ↵ currentCellID << "\t" << mMitoticMode << "\n";
383 }
384
385 void HeCellCycleModel::EnableSequenceSampler()
386 {
387     mSequenceSampler = true;
388     boost::shared_ptr<AbstractCellProperty> p_label_type =
389
390         ↵ mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry()->Get<C
391     mpCell->AddCellProperty(p_label_type);
392 }
393
394

```

```

389 void
400   → HeCellCycleModel::PassDebugWriter(boost::shared_ptr<ColumnDataWriter>
401   → debugWriter, int timeID,
402                                         std::vector<int> varIDs)
403 {
404   mDebug = true;
405   mDebugWriter = debugWriter;
406   mTimeID = timeID;
407   mVarIDs = varIDs;
408 }
409
410 void
411   → HeCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
412   → debugWriter)
413 {
414   mDebug = true;
415   mDebugWriter = debugWriter;
416
417   mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");
418
419   mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
420   mVarIDs.push_back(mDebugWriter->DefineVariable("TiL", "h"));
421   mVarIDs.push_back(mDebugWriter->DefineVariable("CycleDuration",
422     → "h"));
423   mVarIDs.push_back(mDebugWriter->DefineVariable("Phase2Boundary",
424     → "h"));
425   mVarIDs.push_back(mDebugWriter->DefineVariable("Phase3Boundary",
426     → "h"));
427   mVarIDs.push_back(mDebugWriter->DefineVariable("Phase", "No"));
428   mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticModeRV",
429     → "Percentile"));
430   mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode",
431     → "Mode"));
432   mVarIDs.push_back(mDebugWriter->DefineVariable("Label", "binary"));
433
434   mDebugWriter->EndDefineMode();
435 }
436
437 void HeCellCycleModel::WriteDebugData(double currentTiL, unsigned phase,
438   → double mitoticModeRV)
439 {
440   double currentTime = SimulationTime::Instance()->GetTime();
441   double currentCellID = mpCell->GetCellId();

```

```
422     unsigned label = 0;
423     if (mpCell->HasCellProperty<CellLabel>()) label = 1;
424
425     mDebugWriter->PutVariable(mTimeID, currentTime);
426     mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
427     mDebugWriter->PutVariable(mVarIDs[1], currentTiL);
428     mDebugWriter->PutVariable(mVarIDs[2], mCellCycleDuration);
429     mDebugWriter->PutVariable(mVarIDs[3], mMitoticModePhase2);
430     mDebugWriter->PutVariable(mVarIDs[4], mMitoticModePhase3);
431     mDebugWriter->PutVariable(mVarIDs[5], phase);
432     if (!mDeterministic)
433     {
434         mDebugWriter->PutVariable(mVarIDs[6], mitoticModeRV);
435     }
436     mDebugWriter->PutVariable(mVarIDs[7], mMitoticMode);
437     if (mSequenceSampler)
438     {
439         mDebugWriter->PutVariable(mVarIDs[8], label);
440     }
441     mDebugWriter->AdvanceAlongUnlimitedDimension();
442 }
443
444 /*****
445 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
446 *****/
447
448 double HeCellCycleModel::GetAverageTransitCellCycleTime()
449 {
450     return (0.0);
451 }
452
453 double HeCellCycleModel::GetAverageStemCellCycleTime()
454 {
455     return (0.0);
456 }
457
458 void HeCellCycleModel::OutputCellCycleModelParameters(out_stream&
459             rParamsFile)
460 {
461     *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
462             "</CellCycleDuration>\n";
463
464     // Call method on direct parent class
```

```

463     ↳ AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
464 }
465
466 // Serialization for Boost ≥ 1.36
467 #include "SerializationExportWrapperForCpp.hpp"
468 CHASTE_CLASS_EXPORT(HeCellCycleModel)

```

15.1.28 /src/HeCellCycleModel.hpp

```

1 #ifndef HECELLCYCLEMODEL_HPP_
2 #define HECELLCYCLEMODEL_HPP_
3
4 #include "AbstractSimpleCellCycleModel.hpp"
5 #include "RandomNumberGenerator.hpp"
6 #include "Cell.hpp"
7 #include "TransitCellProliferativeType.hpp"
8 #include "DifferentiatedCellProliferativeType.hpp"
9 #include "SmartPointers.hpp"
10 #include "ColumnDataWriter.hpp"
11 #include "LogFile.hpp"
12 #include "CellLabel.hpp"
13 #include "HeAth5Mo.hpp"
14
15 /*****
16 * HE CELL CYCLE MODEL
17 * As described in He et al. 2012 [He2012] doi:
18 *   ↳ 10.1016/j.neuron.2012.06.033
19 *
20 * USE: By default, HeCellCycleModels are constructed with the parameter
21 *   ↳ fit reported in [He2012].
22 * In normal use, the model steps through three phases of mitotic mode
23 *   ↳ probability parameterisation.
24 * PP = symmetric proliferative mitotic mode, both progeny remain mitotic
25 * PD = asymmetric proliferative mitotic mode, one progeny exits the cell
26 *   ↳ cycle and differentiates
27 * DD = symmetric differentiative mitotic mode, both progeny exit the
28 *   ↳ cell cycle and differentiate
29 *
30 * Change default model parameters with SetModelParameters(<params>);
31 * Enable deterministic model alternative with
32 *   ↳ EnableDeterministicMode(<params>);

```

```
27  *
28  * 2 per-model-event output modes:
29  * EnableModeEventOutput() enables mitotic mode event logging-all cells
30  *   ↳ will write to the singleton log file
31  * EnableModelDebugOutput() enables more detailed debug output, each seed
32  *   ↳ will have its own file written to
33  * by a ColumnDataWriter passed to it from the test
34  * (eg. by the SetupDebugOutput helper function in the project simulator)
35  *
36  *
37  *****/
38
39 class HeCellCycleModel : public AbstractSimpleCellCycleModel
40 {
41     friend class TestSimpleCellCycleModels;
42
43 private:
44
45     /** Needed for serialization.*/
46     friend class boost::serialization::access;
47
48     * Archive the cell-cycle model and random number generator, never
49     * used directly - boost uses this.
50     *
51     * @param archive
52     * @param version the current version of this class
53     */
54     template<class Archive>
55     void serialize(Archive & archive, const unsigned int version)
56     {
57         archive &
58             ↳ boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);
59
60         SerializableSingleton<RandomNumberGenerator>* p_wrapper =
61
62             ↳ RandomNumberGenerator::Instance()→GetSerializationWrapper();
63         archive & p_wrapper;
64         archive & mCellCycleDuration;
65     }
66 }
```

```
63 //Private write functions for models
64 void WriteModeEventOutput();
65 void WriteDebugData(double currTiL, unsigned phase, double
66 → percentile);
67
68 protected:
69     //mode/output variables
70     bool mKillSpecified;
71     bool mDeterministic;
72     bool mOutput;
73     double mEventStartTime;
74     bool mSequenceSampler;
75     bool mSeqSamplerLabelSister;
76     //debug writer stuff
77     bool mDebug;
78     int mTimeID;
79     std::vector<int> mVarIDs;
80     boost::shared_ptr<ColumnDataWriter> mDebugWriter;
81     //model parameters and state memory vars
82     double mTiLOffset;
83     double mGammaShift;
84     double mGammaShape;
85     double mGammaScale;
86     double mSisterShiftWidth;
87     double mMitoticModePhase2;
88     double mMitoticModePhase3;
89     double mPhaseShiftWidth;
90     double mPhase1PP;
91     double mPhase1PD;
92     double mPhase2PP;
93     double mPhase2PD;
94     double mPhase3PP;
95     double mPhase3PD;
96     unsigned mMitoticMode;
97     unsigned mSeed;
98     bool mTimeDependentCycleDuration;
99     double mPeakRateTime;
100    double mIncreasingRateSlope;
101    double mDecreasingRateSlope;
102    double mBaseGammaScale;
103
104 /**
```

```
105     * Protected copy-constructor for use by CreateCellCycleModel().  
106     *  
107     * The only way for external code to create a copy of a cell cycle  
108     * model  
109     * is by calling that method, to ensure that a model of the correct  
110     * subclass is created.  
111     * This copy-constructor helps subclasses to ensure that all member  
112     * variables are correctly copied when this happens.  
113     *  
114     * This method is called by child classes to set member variables for  
115     * a daughter cell upon cell division.  
116     * Note that the parent cell cycle model will have had  
117     * ResetForDivision() called just before CreateCellCycleModel() is  
118     * called,  
119     * so performing an exact copy of the parent is suitable behaviour.  
120     * Any daughter-cell-specific initialisation  
121     * can be done in InitialiseDaughterCell().  
122     *  
123     * @param rModel the cell cycle model to copy.  
124     */  
125 HeCellCycleModel(const HeCellCycleModel& rModel);  
126  
127 public:  
128  
129     /**  
130     * Constructor - just a default, mBirthTime is set in the  
131     * AbstractCellCycleModel class.  
132     */  
133 HeCellCycleModel();  
134  
135     /**  
136     * SetCellCycleDuration() method to set length of cell cycle  
137     */  
138 void SetCellCycleDuration();  
139  
140     /**  
141     * Overridden builder method to create new copies of  
142     * this cell-cycle model.  
143     *  
144     * @return new cell-cycle model  
145     */  
146 AbstractCellCycleModel* CreateCellCycleModel();  
147  
148
```

```

140  /**
141   * Overridden ResetForDivision() method.
142   * Contains general mitotic mode logic
143   */
144  void ResetForDivision();

145
146  /**
147   * Overridden Initialise() method
148   * Used to give an appropriate mCellCycleDuration to cells w/ TiL
149   * offsets
150   * sets mReadytoDivide to false as appropriate
151   * Initialises as transit proliferative type
152   */
153  void Initialise();

154  /**
155   * Overridden InitialiseDaughterCell() method.
156   * Used to apply sister-cell time shifting (cell cycle duration,
157   * deterministic phase boundaries)
158   * Used to implement asymmetric mitotic mode
159   */
160  void InitialiseDaughterCell();

161  /*Model setup functions for standard He (SetModelParameters) and
162   * deterministic alternative (SetDeterministicMode) models
163   * Default parameters are from refits of He et al + deterministic
164   * alternatives
165   * He 2012 params: mitoticModePhase2 = 8, mitoticModePhase3 = 15,
166   * p1PP = 1, p1PD = 0, p2PP = .2, p2PD = .4, p3PP = .2, p3PD = 0
167   * gammaShift = 4, gammaShape = 2, gammaScale = 1, sisterShift = 1
168   */
169  void SetModelParameters(double tiLOffset = 0, double
170   * mitoticModePhase2 = 8, double mitoticModePhase3 = 15,
171   * double phase1PP = 1, double phase1PD = 0,
172   * double phase2PP = .2, double phase2PD =
173   * .4,
174   * double phase3PP = .2, double phase3PD = 0,
175   * double gammaShift = 4, double gammaShape
176   * = 2,
177   * double gammaScale = 1, double sisterShift =
178   * 1);
179  void SetDeterministicMode(double tiLOffset = 0, double
180   * mitoticModePhase2 = 8, double mitoticModePhase3 = 15,

```

```

171         double phaseShiftWidth = 1, double
172             ↵ gammaShift = 4, double gammaShape = 2,
173             double gammaScale = 1, double sisterShift =
174                 ↵ 1);
175     void SetTimeDependentCycleDuration(double peakRateTime, double
176         ↵ increasingSlope, double decreasingSlope);
177
178     //Function to set mKillSpecified = true; marks specified neurons for
179         ↵ death and removal from population
180     //Intended to help w/ resource consumption for WanSimulator
181     void EnableKillSpecified();
182
183     //Functions to enable per-cell mitotic mode logging for mode rate &
184         ↵ sequence sampling fixtures
185     //Uses singleton logfile
186     void EnableModeEventOutput(double eventStart, unsigned seed);
187     void EnableSequenceSampler();
188
189     //More detailed debug output. Needs a ColumnDataWriter passed to it
190     //Only declare ColumnDataWriter directory, filename, etc; do not set
191         ↵ up otherwise
192     //Use PassDebugWriter if the writer is already enabled elsewhere (ie.
193         ↵ in a Wan stem cell cycle model)
194     void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
195         ↵ debugWriter);
196     void PassDebugWriter(boost::shared_ptr<ColumnDataWriter> debugWriter,
197         ↵ int timeID, std::vector<int> varIDs);
198
199     //Not used, but must be overwritten lest HeCellCycleModels be
200         ↵ abstract
201     double GetAverageTransitCellCycleTime();
202     double GetAverageStemCellCycleTime();
203
204     /**
205      * Overridden OutputCellCycleModelParameters() method.
206      *
207      * @param rParamsFile the file stream to which the parameters are
208          ↵ output
209      */
210     virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
211 };
212
213 #include "SerializationExportWrapper.hpp"

```

```

203 // Declare identifier for the serializer
204 CHASTE_CLASS_EXPORT(HeCellCycleModel)
205
206 #endif /*HECELLCYCLEMODEL_HPP_*/

```

15.1.29 /src/OffLatticeSimulationPropertyStop.cpp

```

1 #include "OffLatticeSimulationPropertyStop.hpp"
2
3 #include <boost/make_shared.hpp>
4
5 #include "CellBasedEventHandler.hpp"
6 #include "ForwardEulerNumericalMethod.hpp"
7 #include "StepSizeException.hpp"
8
9 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
10 OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::OffLatticeSimulationPropertyStop(
11     rCellPopulation,
12     bool deleteCellPopulationInDestructor,
13     bool initialiseCells
14 ) :
15     AbstractCellBasedSimulation<ELEMENT_DIM,SPACE_DIM>(rCellPopulation,
16     deleteCellPopulationInDestructor, initialiseCells),
17     p_property()
18 {
19     if (
20         !dynamic_cast<AbstractOffLatticeCellPopulation<ELEMENT_DIM,SPACE_DIM>>(&rCellPopulation)
21     ) {
22         EXCEPTION("OffLatticeSimulations require a subclass of
23             AbstractOffLatticeCellPopulation.");
24     }
25 }
26
27 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
28 bool OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::StoppingEventHasOccurred()
29 {
30     if(p_property->GetCellCount()<1){
31         return true;
32     }
33     else{

```

```
30             return false;
31         }
32     }
33
34 //Public access to Stopping Event bool
35 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
36 bool
37     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::HasStoppingEventOccurred
38 {
39     return StoppingEventHasOccurred();
40 }
41 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
42 void
43     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::SetStopProperty(boost::shared_p
44     → stopPropertySetting)
45 {
46     p_property = stopPropertySetting;
47 }
48 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
49 void
50     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::AddForce(boost::shared_p
51     → pForce)
52 {
53     mForceCollection.push_back(pForce);
54 }
55 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
56 void
57     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RemoveAllForces()
58 {
59     mForceCollection.clear();
60 }
61 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
62 void
63     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::AddCellPopulationBoundaryC
64     → pBoundaryCondition)
65 {
66     mBoundaryConditions.push_back(pBoundaryCondition);
67 }
```

```

65 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
66 void
67   → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RemoveAllCellPopulationB
68 {
69   mBoundaryConditions.clear();
70 }
71
72 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
73 void
74   → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::SetNumericalMethod(boost
75   → SPACE_DIM> > pNumericalMethod)
76 {
77   mpNumericalMethod = pNumericalMethod;
78 }
79
80 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
81 const boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM, SPACE_DIM> >
82   → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::GetNumericalMethod()
83   → const
84 {
85   return mpNumericalMethod;
86 }
87
88 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
89 const std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM>
90   → > >&
91   → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::rGetForceCollection()
92   → const
93 {
94   return mForceCollection;
95 }
96
97 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
98 void
99   → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::UpdateCellLocationsAndTo
100 {
101   CellBasedEventHandler::BeginEvent(CellBasedEventHandler::POSITION);
102
103   double time_advanced_so_far = 0;
104   double target_time_step = this→mDt;
105   double present_time_step = this→mDt;
106
107   while (time_advanced_so_far < target_time_step)
108 }
```

```

99  {
100     // Store the initial node positions (these may be needed when
101     // applying boundary conditions)
102     std::map<Node<SPACE_DIM>*, c_vector<double, SPACE_DIM> >
103     old_node_locations;
104
105     for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator
106         node_iter =
107         this->mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
108         node_iter !=
109         this->mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
110         ++node_iter)
111     {
112         old_node_locations[&(*node_iter)] =
113             (node_iter)->rGetLocation();
114     }
115
116     // Try to update node positions according to the numerical method
117     try
118     {
119         mpNumericalMethod->UpdateAllNodePositions(present_time_step);
120         ApplyBoundaries(old_node_locations);
121
122         // Successful time step! Update time_advanced_so_far
123         time_advanced_so_far += present_time_step;
124
125         // If using adaptive timestep, then increase the
126         // present_time_step (by 1% for now)
127         if (mpNumericalMethod->HasAdaptiveTimestep())
128         {
129             ///\todo #2087 Make this a settable member variable
130             double timestep_increase = 0.01;
131             present_time_step =
132                 std::min((1+timestep_increase)*present_time_step,
133                         target_time_step - time_advanced_so_far);
134         }
135
136     }
137     catch (StepSizeException& e)
138     {
139         // Detects if a node has travelled too far in a single time
140         // step
141         if (mpNumericalMethod->HasAdaptiveTimestep())

```

```

132     {
133         // If adaptivity is switched on, revert node locations
134         → and choose a suitably smaller time step
135         RevertToOldLocations(old_node_locations);
136         present_time_step = std::min(e.GetSuggestedNewStep(),
137             → target_time_step - time_advanced_so_far);
138     }
139     else
140     {
141         // If adaptivity is switched off, terminate with an error
142         EXCEPTION(e.what());
143     }
144
145     CellBasedEventHandler::EndEvent(CellBasedEventHandler::POSITION);
146 }
147
148 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
149 void
150     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RevertToOldLocations(std::
151         → c_vector<double, SPACE_DIM> > oldNodeLoctions)
152 {
153     for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator
154         → node_iter =
155         → this→mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
156         node_iter ≠
157         → this→mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
158         ++node_iter)
159     {
160         (node_iter)→rGetModifiableLocation() =
161             → oldNodeLoctions[&(*node_iter)];
162     }
163 }
164
165 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
166 void
167     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::ApplyBoundaries(std::map
168         → SPACE_DIM> > oldNodeLoctions)
169 {
170     // Apply any boundary conditions

```

```

163   for (typename
164     std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT>>::iterator bcs_iter = mBoundaryConditions.begin();
165     bcs_iter != mBoundaryConditions.end();
166     ++bcs_iter)
167   {
168     (*bcs_iter)->ImposeBoundaryCondition(oldNodeLoctions);
169   }
170
171   // Verify that each boundary condition is now satisfied
172   for (typename
173     std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT>>::iterator bcs_iter = mBoundaryConditions.begin();
174     bcs_iter != mBoundaryConditions.end();
175     ++bcs_iter)
176   {
177     if (!(*bcs_iter)->VerifyBoundaryCondition())
178     {
179       EXCEPTION("The cell population boundary conditions are
180       incompatible.");
181     }
182   }
183
184 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
185 void
186   OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::WriteVisualizerSetupFile()
187 {
188   if (PetscTools::AmMaster())
189   {
190     for (unsigned i=0; i<this->mForceCollection.size(); i++)
191     {
192       this->mForceCollection[i]->WriteDataToVisualizerSetupFile(this->mpViz);
193     }
194   }
195
196 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
```

```

197 void
198 {
199     // Clear all forces
200     for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator
201         node_iter =
202         this->mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
203         node_iter !=
204         this->mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
205         ++node_iter)
206     {
207         node_iter->ClearAppliedForce();
208     }
209
210     // Use a forward Euler method by default, unless a numerical method
211     // has been specified already
212     if (mpNumericalMethod == nullptr)
213     {
214         mpNumericalMethod =
215             boost::make_shared<ForwardEulerNumericalMethod<ELEMENT_DIM,
216             SPACE_DIM>>();
217
218         mpNumericalMethod->SetCellPopulation(dynamic_cast<AbstractOffLatticeCellPopul
219         mpNumericalMethod->SetForceCollection(&mForceCollection);
220     }
221
222     mpNumericalMethod->SetupSolve();
223
224     OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM>::OutputAdditionalSimulati
225     rParamsFile)
226
227     // Loop over forces
228     *rParamsFile << "\n\t<Forces>\n";
229     for (typename
230         std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM>
231         >>::iterator iter = mForceCollection.begin();
232         iter != mForceCollection.end();
233         ++iter)
234     {
235         // Output force details
236         (*iter)->OutputForceInfo(rParamsFile);
237     }

```

```

228     *rParamsFile << "\t</Forces>\n";
229
230     // Loop over cell population boundary conditions
231     *rParamsFile << "\n\t<CellPopulationBoundaryConditions>\n";
232     for (typename
233         → std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT>
234         → >>::iterator iter = mBoundaryConditions.begin();
235         iter ≠ mBoundaryConditions.end();
236         ++iter)
237     {
238         // Output cell boundary condition details
239         (*iter)→OutputCellPopulationBoundaryConditionInfo(rParamsFile);
240     }
241     *rParamsFile << "\t</CellPopulationBoundaryConditions>\n";
242
243     // Output numerical method details
244     *rParamsFile << "\n\t<NumericalMethod>\n";
245     mpNumericalMethod→OutputNumericalMethodInfo(rParamsFile);
246     *rParamsFile << "\t</NumericalMethod>\n";
247 }
248
249 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
250 void
251     → OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::OutputSimulationParameter
252     → rParamsFile)
253 {
254     // No new parameters to output, so just call method on direct parent
255     → class
256
257     → AbstractCellBasedSimulation<ELEMENT_DIM,SPACE_DIM>::OutputSimulationParameter
258 }
259
260 // Explicit instantiation
261 template class OffLatticeSimulationPropertyStop<1,1>;
262 template class OffLatticeSimulationPropertyStop<1,2>;
263 template class OffLatticeSimulationPropertyStop<2,2>;
264 template class OffLatticeSimulationPropertyStop<1,3>;
265 template class OffLatticeSimulationPropertyStop<2,3>;
266 template class OffLatticeSimulationPropertyStop<3,3>;
267
268 // Serialization for Boost ≥ 1.36
269 #include "SerializationExportWrapperForCpp.hpp"
270 EXPORT_TEMPLATE_CLASS_ALL_DIMS(OffLatticeSimulationPropertyStop)

```

15.1.30 /src/OffLatticeSimulationPropertyStop.hpp

```

1 #ifndef OFFLATTICESIMULATIONPROPERTYSTOP_HPP_
2 #define OFFLATTICESIMULATIONPROPERTYSTOP_HPP_
3
4 #include "AbstractCellBasedSimulation.hpp"
5 #include "AbstractForce.hpp"
6 #include "AbstractCellPopulationBoundaryCondition.hpp"
7 #include "AbstractNumericalMethod.hpp"
8
9 #include "ChasteSerialization.hpp"
10 #include <boost/serialization/base_object.hpp>
11 #include <boost/serialization/set.hpp>
12 #include <boost/serialization/vector.hpp>
13
14 /**
15  * Run an off-lattice 2D or 3D cell-based simulation using an off-lattice
16  * cell population.
17  *
18  * In cell-centre-based cell populations, each cell is represented by a
19  * single node (corresponding to its centre), and connectivity is defined
20  * either by a Delaunay triangulation or a radius of influence. In
21  * vertex-
22  * based cell populations, each cell is represented by a polytope
23  * (corresponding to its membrane) with a variable number of vertices.
24  * Alternative cell populations may be defined by the user.
25  *
26  * The OffLatticeSimulation is constructed with a CellPopulation, which
27  * updates the correspondence between each Cell and its spatial
28  * representation
29  * and handles cell division (governed by the CellCycleModel associated
30  * with each cell). Once constructed, one or more Force laws may be
31  * passed
32  * to the OffLatticeSimulation object, to define the mechanical
33  * properties
34  * of the CellPopulation. Similarly, one or more CellKillers may be
35  * passed
36  * to the OffLatticeSimulation object to specify conditions in which
37  * Cells
38  * may die, and one or more CellPopulationBoundaryConditions to specify
39  * regions in space beyond which Cells may not move.

```

```

34  */
35 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM = ELEMENT_DIM>
36 class OffLatticeSimulationPropertyStop : public
37   AbstractCellBasedSimulation<ELEMENT_DIM, SPACE_DIM>
38 {
39
40   /** Needed for serialization. */
41   friend class boost::serialization::access;
42   friend class TestOffLatticeSimulation;
43   friend class TestOffLatticeSimulationWithNodeBasedCellPopulation;
44
45 /**
46  * Save or restore the simulation.
47  *
48  * @param archive the archive
49  * @param version the current version of this class
50  */
51 template<class Archive>
52 void serialize(Archive & archive, const unsigned int version)
53 {
54   archive &
55     boost::serialization::base_object<AbstractCellBasedSimulation<ELEMENT_DIM,
56     SPACE_DIM>>(*this);
57   archive & mForceCollection;
58   archive & mBoundaryConditions;
59   archive & mpNumericalMethod;
60 }
61
62 protected:
63   boost::shared_ptr<AbstractCellProperty> p_property;
64   /** The mechanics used to determine the new location of the cells, a
65    * list of the forces. */
66   std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM>>
67     > mForceCollection;
68
69   /** List of boundary conditions. */
70
71   std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_
72     SPACE_DIM>>> mBoundaryConditions;
73
74   /** The numerical method to use in this simulation. Defaults to the
75    * explicit forward Euler method. */
76

```

```
69     boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM, SPACE_DIM>>
70     mpNumericalMethod;
71
72     /**
73      * Overridden UpdateCellLocationsAndTopology() method.
74      *
75      * Calculate forces and update node positions.
76      */
77     virtual void UpdateCellLocationsAndTopology();
78
79     /**
80      * Sends nodes back to the positions given in the input map. Used
81      * after a failed step
82      * when adaptivity is turned on.
83      *
84      * @param oldNodeLoctions A map linking nodes to their old positions.
85      */
86     void RevertToOldLocations(std::map<Node<SPACE_DIM>*, c_vector<double,
87     SPACE_DIM> > oldNodeLoctions);
88
89     /**
90      * Applies any boundary conditions.
91      *
92      * @param oldNodeLoctions Mapping between node indices and old node
93      * locations
94      */
95     void ApplyBoundaries(std::map<Node<SPACE_DIM>*, c_vector<double,
96     SPACE_DIM> > oldNodeLoctions);
97
98     /**
99      * Overridden SetupSolve() method to clear the forces applied to the
100     * nodes.
101     */
102    virtual void SetupSolve();
103
104    /**
105     * Overridden WriteVisualizerSetupFile() method.
106     */
107    virtual void WriteVisualizerSetupFile();
108
109    bool StoppingEventHasOccurred();
110
111 public:
```

```

106
107  /**
108   * Constructor.
109   *
110  * @param rCellPopulation Reference to a cell population object
111  * @param deleteCellPopulationInDestructor Whether to delete the cell
112  * population on destruction to
113  *     * free up memory (defaults to false)
114  * @param initialiseCells Whether to initialise cells (defaults to
115  * true, set to false when loading
116  *     * from an archive)
117  */
118 OffLatticeSimulationPropertyStop(AbstractCellPopulation<ELEMENT_DIM,
119  * SPACE_DIM>& rCellPopulation,
120  *                                     bool
121  *                                     * deleteCellPopulationInDestructor
122  *                                     * = false, bool initialiseCells =
123  *                                     * true);
124
125
126
127
128 void SetStopProperty(boost::shared_ptr<AbstractCellProperty>
129  * stopPropertySetting);
130
131
132
133
134
135
136
137
138

```

```
139     */
140 void AddCellPopulationBoundaryCondition(
141     → boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_DIM,
142     → SPACE_DIM> > pBoundaryCondition);
143 /**
144 * Method to remove all the cell population boundary conditions
145 */
146 void RemoveAllCellPopulationBoundaryConditions();
147 /**
148 * Set the numerical method to be used in this simulation (use this
149 → to solve the mechanics system).
150 *
151 * @param pNumericalMethod pointer to a numerical method object
152 */
153 void
154     → SetNumericalMethod(boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM,
155     → SPACE_DIM> > pNumericalMethod);
156 /**
157 * @return the current numerical method.
158 */
159 const boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM,
160     → SPACE_DIM> > GetNumericalMethod() const;
161 /**
162 * Overridden OutputAdditionalSimulationSetup() method.
163 *
164 * Output any force, boundary condition or numerical method
165 → information.
166 *
167 * @param rParamsFile the file stream to which the parameters are
168 → output
169 */
170 void OutputAdditionalSimulationSetup(out_stream& rParamsFile);
171 /**
172 * Overridden OutputSimulationParameters() method.
173 *
174 * @param rParamsFile the file stream to which the parameters are
175 → output
```

```
173     */
174     virtual void OutputSimulationParameters(out_stream& rParamsFile);
175
176     /**
177      * Directly access the forces attached to this simulation, to allow
178      * their manipulation after archiving.
179      *
180      * @return mForceCollection the vector of pointers to forces attached
181      * to this simulation
182      */
183     const std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM,
184     >>& rGetForceCollection() const;
185 };
186
187 // Serialization for Boost ≥ 1.36
188 #include "SerializationExportWrapper.hpp"
189 EXPORT_TEMPLATE_CLASS_ALL_DIMS(OffLatticeSimulationPropertyStop)
190
191 namespace boost
192 {
193 namespace serialization
194 {
195 /**
196  * Serialize information required to construct an OffLatticeSimulation.
197 */
198 template<class Archive, unsigned ELEMENT_DIM, unsigned SPACE_DIM>
199 inline void save_construct_data(Archive & ar, const
200     OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM> * t,
201                                         const unsigned int file_version)
202 {
203     // Save data required to construct instance
204     const AbstractCellPopulation<ELEMENT_DIM, SPACE_DIM>*
205         p_cell_population = &(t->rGetCellPopulation());
206     ar & p_cell_population;
207 }
208
209 /**
210  * De-serialize constructor parameters and initialise an
211  * OffLatticeSimulation.
212 */
213 template<class Archive, unsigned ELEMENT_DIM, unsigned SPACE_DIM>
214 inline void load_construct_data(Archive & ar,
215     OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM> * t,
```

```

209                               const unsigned int file_version)
210 {
211     // Retrieve data from archive required to construct new instance
212     AbstractCellPopulation<ELEMENT_DIM, SPACE_DIM>* p_cell_population;
213     ar >> p_cell_population;
214
215     // Invoke inplace constructor to initialise instance, middle two
216     // variables set extra
217     // member variables to be deleted as they are loaded from archive and
218     // to not initialise cells.
219     ::new (t) OffLatticeSimulationPropertyStop<ELEMENT_DIM,
220           SPACE_DIM>(*p_cell_population, true, false);
221 }
222 }
223 } // namespace
224
225 #endif /*OFFLATTICESIMULATIONPROPERTYSTOP_HPP_*/

```

15.1.31 /src/WanStemCellCycleModel.cpp

```

1 #include "WanStemCellCycleModel.hpp"
2
3 WanStemCellCycleModel::WanStemCellCycleModel() :
4     AbstractSimpleCellCycleModel(), mExpandingStemPopulation(false),
5     mPopulation(), mOutput(false), mEventStartTime(
6         72.0), mDebug(false), mTimeID(), mVarIDs(),
7         mDebugWriter(), mBasePopulation(), mGammaShift(4.0),
8         mGammaShape(
9             2.0), mGammaScale(1.0), mMitoticMode(0), mSeed(0),
10        mTimeDependentCycleDuration(false), mPeakRateTime(),
11        mIncreasingRateSlope(), mDecreasingRateSlope(),
12        mBaseGammaScale(), mHeParamVector(
13            { 8, 15, 1, 0, .2, .4, .2, 0, 4, 2, 1, 1 })
14    {
15    }
16
17 WanStemCellCycleModel::WanStemCellCycleModel(const WanStemCellCycleModel&
18      rModel) :
19     AbstractSimpleCellCycleModel(rModel),
20     mExpandingStemPopulation(rModel.mExpandingStemPopulation),
21     mPopulation(
22

```

```

13         rModel.mPopulation), mOutput(rModel.mOutput),
14         ↳ mEventStartTime(rModel.mEventStartTime), mDebug(
15             rModel.mDebug), mTimeID(rModel.mTimeID),
16             ↳ mVarIDs(rModel.mVarIDs),
17             ↳ mDebugWriter(rModel.mDebugWriter), mBasePopulation(
18                 rModel.mBasePopulation), mGammaShift(rModel.mGammaShift),
19                 ↳ mGammaShape(rModel.mGammaShape), mGammaScale(
20                     rModel.mGammaScale), mMitoticMode(rModel.mMitoticMode),
21                     ↳ mSeed(rModel.mSeed), mTimeDependentCycleDuration(
22                         rModel.mTimeDependentCycleDuration),
23                         ↳ mPeakRateTime(rModel.mPeakRateTime),
24                         ↳ mIncreasingRateSlope(
25                             rModel.mIncreasingRateSlope),
26                             ↳ mDecreasingRateSlope(rModel.mDecreasingRateSlope),
27                             ↳ mBaseGammaScale(
28                                 rModel.mBaseGammaScale),
29                                 ↳ mHeParamVector(rModel.mHeParamVector)
30     {
31 }
32
33 AbstractCellCycleModel* WanStemCellCycleModel::CreateCellCycleModel()
34 {
35     return new WanStemCellCycleModel(*this);
36 }
37
38 void WanStemCellCycleModel::SetCellCycleDuration()
39 {
40     RandomNumberGenerator* p_random_number_generator =
41         ↳ RandomNumberGenerator::Instance();
42
43     mCellCycleDuration = mGammaShift +
44         ↳ p_random_number_generator->GammaRandomDeviate(mGammaShape,
45             ↳ mGammaScale);
46
47     /*****
48     * CELL CYCLE DURATION RANDOM VARIABLE
49     *****/
50
51 //Wan stem cell cycle length determined by the same formula as He
52     ↳ RPCs
53 /*
54     if (!mTimeDependentCycleDuration) //Normal operation, cell cycle
55         ↳ length stays constant
56     {

```

```

41     //He cell cycle length determined by shifted gamma distribution
42     → reflecting 4 hr refractory period followed by gamma pdf
43     mCellCycleDuration = mGammaShift +
44     → p_random_number_generator→GammaRandomDeviate(mGammaShape,
45     → mGammaScale);
46     }*/
47
48 /**
49 * Variable cycle length
50 * Give -ve mIncreasingRateSlope and +ve mDecreasingRateSlope,
51 * cell cycle length linearly declines (increasing rate), then
52 * increases, switching at mPeakRateTime
53 */
54
55 /*
56 else
57 {
58 double currTime = SimulationTime::Instance()→GetTime();
59 if (currTime ≤ mPeakRateTime)
60 {
61     mGammaScale = std::max((mBaseGammaScale - currTime *
62     → mIncreasingRateSlope), .000000000001);
63     if (currTime > mPeakRateTime)
64     {
65         mGammaScale = std::max(((mBaseGammaScale - mPeakRateTime *
66     → mIncreasingRateSlope)
67     + (mBaseGammaScale + (currTime - mPeakRateTime) *
68     → mDecreasingRateSlope)),.000000000001);
69     }
70     Timer::Print("mGammaScale: " + std::to_string(mGammaScale));
71     mCellCycleDuration = mGammaShift +
72     → p_random_number_generator→GammaRandomDeviate(mGammaShape,
73     → mGammaScale);
74     */
75 }
76 */
77 }

78 void WanStemCellCycleModel::ResetForDivision()
79 {
80
81     mMitoticMode = 1; //by default, asymmetric division giving rise to He
82     → cell (mode 1)

```

```

74     if (mExpandingStemPopulation)
75     {
76         double currRetinaAge = SimulationTime::Instance()→GetTime() +
77             → mEventStartTime;
78         double lensGrowthFactor = .09256 * pow(currRetinaAge, .52728); // //
79         unsigned currentPopulationTarget = int(std::round(mBasePopulation
80             → * lensGrowthFactor));
81
82         unsigned currentStemPopulation =
83             (mPopulation→GetCellProliferativeTypeCount())[0];
84         if (currentStemPopulation < currentPopulationTarget)
85         {
86             mMitoticMode = 0; //if the current population is < target,
87             → symmetrical stem-stem division occurs (mode 0)
88         }
89     }
90
91     /*****
92     * Write mitotic event to relevant files
93     * *****/
94     if (mDebug)
95     {
96         WriteDebugData();
97     }
98
99     if (mOutput)
100    {
101        WriteModeEventOutput();
102    }
103
104 void WanStemCellCycleModel::Initialise()
105 {
106
107     boost::shared_ptr<AbstractCellProperty> p_Stem =
108
109         → mpCell→rGetCellPropertyCollection().GetCellPropertyRegistry()→Get<S
110     mpCell→SetCellProliferativeType(p_Stem);

```

```

111     SetCellCycleDuration();
112 }
113
114 void WanStemCellCycleModel::InitialiseDaughterCell()
115 {
116
117     if (mMitoticMode == 1)
118     {
119         /*****  

120          * RPC-fated cells are given HeCellCycleModel  

121          *****/  

122
123     double tiLOffset = -(SimulationTime::Instance()→GetTime());
124     //Initialise a HeCellCycleModel and set it up with appropriate
125     // TiL value & parameters
126     HeCellCycleModel* p_cycle_model = new HeCellCycleModel;
127     p_cycle_model→SetModelParameters(tiLOffset, mHeParamVector[0],
128                                     mHeParamVector[1], mHeParamVector[2],
129                                     mHeParamVector[3],
130                                     mHeParamVector[4],
131                                     mHeParamVector[5],
132                                     mHeParamVector[6],
133                                     mHeParamVector[7],
134                                     mHeParamVector[8],
135                                     mHeParamVector[9],
136                                     mHeParamVector[10],
137                                     mHeParamVector[11]);
138     p_cycle_model→EnableKillSpecified();
139
140     //if debug output is enabled for the stem cell, enable it for its
141     // progenitor offspring
142     if (mDebug)
143     {
144         p_cycle_model→PassDebugWriter(mDebugWriter, mTimeID,
145                                     mVarIDs);
146     }
147
148     mpCell→SetCellCycleModel(p_cycle_model);
149     p_cycle_model→Initialise();
150 }
151
152 else
153 {

```

```

144         SetCellCycleDuration();
145     }
146 }
147
148 void WanStemCellCycleModel::SetModelParameters(double gammaShift, double
149     ↪ gammaShape, double gammaScale,
150                     std::vector<double>
151     ↪ heParamVector)
152 {
153     mGammaShift = gammaShift;
154     mGammaShape = gammaShape;
155     mGammaScale = gammaScale;
156     mHeParamVector = heParamVector;
157 }
158
159 void WanStemCellCycleModel::EnableExpandingStemPopulation(int
160     ↪ basePopulation,
161                     boost::shared_ptr<AbstractPopulation>
162     ↪ p_population)
163 {
164     mExpandingStemPopulation = true;
165     mBasePopulation = basePopulation;
166     mPopulation = p_population;
167 }
168
169 void WanStemCellCycleModel::SetTimeDependentCycleDuration(double
170     ↪ peakRateTime, double increasingSlope,
171                     double
172     ↪ decreasingSlope)
173 {
174     mTimeDependentCycleDuration = true;
175     mPeakRateTime = peakRateTime;
176     mIncreasingRateSlope = increasingSlope;
177     mDecreasingRateSlope = decreasingSlope;
178     mBaseGammaScale = mGammaScale;
179 }
180
181 void WanStemCellCycleModel::EnableModeEventOutput(double eventStart,
182     ↪ unsigned seed)
183 {
184     mOutput = true;
185     mEventStartTime = eventStart;

```

```

179     mSeed = seed;
180 }
181
182 void WanStemCellCycleModel::WriteModeEventOutput()
183 {
184     double currentTime = SimulationTime::Instance()→GetTime() +
185         → mEventStartTime;
186     CellPtr currentCell = GetCell();
187     double currentCellID = (double) currentCell→GetCellId();
188     (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" <<
189         → currentCellID << "\t" << mMitoticMode << "\n"; //
190 }
191
192 void
193     → WanStemCellCycleModel::EnableModelDebugOutput(boost :: shared_ptr<ColumnDataWriter>
194         → debugWriter)
195 {
196     mDebug = true;
197     mDebugWriter = debugWriter;
198
199     mTimeID = mDebugWriter→DefineUnlimitedDimension("Time", "h");
200     mVarIDs.push_back(mDebugWriter→DefineVariable("CellID", "No"));
201     mVarIDs.push_back(mDebugWriter→DefineVariable("TIL", "h"));
202     mVarIDs.push_back(mDebugWriter→DefineVariable("CycleDuration",
203         → "h"));
204     mVarIDs.push_back(mDebugWriter→DefineVariable("Phase2Boundary",
205         → "h"));
206     mVarIDs.push_back(mDebugWriter→DefineVariable("Phase3Boundary",
207         → "h"));
208     mVarIDs.push_back(mDebugWriter→DefineVariable("Phase", "No"));
209     mVarIDs.push_back(mDebugWriter→DefineVariable("MitoticModeRV",
210         → "Percentile"));
211     mVarIDs.push_back(mDebugWriter→DefineVariable("MitoticMode",
212         → "Mode"));
213     mVarIDs.push_back(mDebugWriter→DefineVariable("Label", "binary"));
214
215     mDebugWriter→EndDefineMode();
216 }
217
218
219 void WanStemCellCycleModel::WriteDebugData()
220 {
221     double currentTime = SimulationTime::Instance()→GetTime();
222     double currentCellID = mpCell→GetCellId();

```

```
213     mDebugWriter→PutVariable(mTimeID, currentTime);
214     mDebugWriter→PutVariable(mVarIDs[0], currentCellID);
215     mDebugWriter→PutVariable(mVarIDs[1], 0);
216     mDebugWriter→PutVariable(mVarIDs[2], mCellCycleDuration);
217     mDebugWriter→PutVariable(mVarIDs[3], 0);
218     mDebugWriter→PutVariable(mVarIDs[4], 0);
219     mDebugWriter→PutVariable(mVarIDs[5], 0);
220     mDebugWriter→PutVariable(mVarIDs[7], mMitoticMode);
221     mDebugWriter→AdvanceAlongUnlimitedDimension();
222 }
223
224 /**
225  * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
226  */
227 ****
228
229 double WanStemCellCycleModel::GetAverageTransitCellCycleTime()
230 {
231     return (0.0);
232 }
233
234 double WanStemCellCycleModel::GetAverageStemCellCycleTime()
235 {
236     return (0.0);
237 }
238
239 void WanStemCellCycleModel::OutputCellCycleModelParameters(out_stream&
240     ↪ rParamsFile)
241 {
242     *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
243     ↪ "</CellCycleDuration>\n";
244
245 // Call method on direct parent class
246
247     ↪ AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
248 }
249
250 // Serialization for Boost ≥ 1.36
251 #include "SerializationExportWrapperForCpp.hpp"
252 CHASTE_CLASS_EXPORT(WanStemCellCycleModel)
```

15.1.32 /src/WanStemCellCycleModel.hpp

```

1 #ifndef WANSTEMCELLCYCLEMODEL_HPP_
2 #define WANSTEMCELLCYCLEMODEL_HPP_
3
4 #include "AbstractCellPopulation.hpp"
5 #include "AbstractSimpleCellCycleModel.hpp"
6 #include "RandomNumberGenerator.hpp"
7 #include "Cell.hpp"
8 #include "StemCellProliferativeType.hpp"
9 #include "SmartPointers.hpp"
10 #include "ColumnDataWriter.hpp"
11 #include "LogFile.hpp"
12
13 #include "HeCellCycleModel.hpp"
14
15 /*****
16  * WAN STEM CELL CYCLE MODEL
17  * A simple model standing in for the CMZ "stem cells" described in'
18  * Wan et al. 2016 [Wan2016]
19  *
20  * USE: By default, WanStemCellCycleModels consistently divide
21  *      ↳ asymmetrically.
22  * InitialiseDaughterCell() marks offspring for RPC fate
23  * So-marked cells are given HeCellCycleModels for their next division.
24  *
25  * Change default model parameters with SetModelParameters(<params>);
26  *
27  * 2 per-model-event output modes:
28  * EnableModeEventOutput() enables mitotic mode event logging-all cells
29  *      ↳ will write to the singleton log file
30  * EnableModelDebugOutput() enables more detailed debug output, each seed
31  *      ↳ will have its own file written to
32  * by a ColumnDataWriter passed to it from the test
33  * (eg. by the SetupDebugOutput helper function in the project simulator)
34  *
35  * 1 mitotic-event-sequence sampler (only samples one "path" through the
36  *      ↳ lineage):
37  * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic
38  *      ↳ event type to a string in the singleton log file
39  *
40 *****/

```

```
37 class WanStemCellCycleModel : public AbstractSimpleCellCycleModel
38 {
39     friend class TestSimpleCellCycleModels;
40
41 private:
42
43     /** Needed for serialization. */
44     friend class boost::serialization::access;
45     /**
46         * Archive the cell-cycle model and random number generator, never
47         * used directly - boost uses this.
48         *
49         * @param archive the archive
50         * @param version the current version of this class
51         */
52     template<class Archive>
53     void serialize(Archive & archive, const unsigned int version)
54     {
55         archive &
56             boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);
57
58         SerializableSingleton<RandomNumberGenerator>* p_wrapper =
59
60             RandomNumberGenerator::Instance()→GetSerializationWrapper();
61         archive & p_wrapper;
62         archive & mCellCycleDuration;
63     }
64
65     //Private write functions for models
66     void WriteModeEventOutput();
67     void WriteDebugData();
68
69 protected:
70     //mode/output variables
71     bool mExpandingStemPopulation;
72     boost::shared_ptr<AbstractCellPopulation<2>> mPopulation;
73     bool mOutput;
74     double mEventStartTime;
75     //debug writer stuff
76     bool mDebug;
77     int mTimeID;
78     std::vector<int> mVarIDs;
79     boost::shared_ptr<ColumnDataWriter> mDebugWriter;
```

```

77     //model parameters and state memory vars
78     int mBasePopulation;
79     double mGammaShift;
80     double mGammaShape;
81     double mGammaScale;
82     unsigned mMitoticMode;
83     unsigned mSeed;
84     bool mTimeDependentCycleDuration;
85     double mPeakRateTime;
86     double mIncreasingRateSlope;
87     double mDecreasingRateSlope;
88     double mBaseGammaScale;
89     std::vector<double> mHeParamVector;

90
91 /**
92  * Protected copy-constructor for use by CreateCellCycleModel().
93  *
94  * The only way for external code to create a copy of a cell cycle
95  * model
96  * is by calling that method, to ensure that a model of the correct
97  * subclass is created.
98  * This copy-constructor helps subclasses to ensure that all member
99  * variables are correctly copied when this happens.
100  *
101  * This method is called by child classes to set member variables for
102  * a daughter cell upon cell division.
103  * Note that the parent cell cycle model will have had
104  * ResetForDivision() called just before CreateCellCycleModel() is
105  * called,
106  * so performing an exact copy of the parent is suitable behaviour.
107  * Any daughter-cell-specific initialisation
108  * can be done in InitialiseDaughterCell().
109  *
110  * @param rModel the cell cycle model to copy.
111  */
112 WanStemCellCycleModel(const WanStemCellCycleModel& rModel);

113
114 public:
115
116 /**
117  * Constructor - just a default, mBirthTime is set in the
118  * AbstractCellCycleModel class.
119  */

```

```
112     WanStemCellCycleModel();  
113  
114     /**  
115      * SetCellCycleDuration() method to set length of cell cycle  
116      */  
117     void SetCellCycleDuration();  
118  
119     /**pro  
120      * Overridden builder method to create new copies of  
121      * this cell-cycle model.  
122      *  
123      * @return new cell-cycle model  
124      */  
125     AbstractCellCycleModel* CreateCellCycleModel();  
126  
127     /**  
128      * Overridden ResetForDivision() method.  
129      ***/  
130     void ResetForDivision();  
131  
132     /**  
133      * Overridden Initialise() method.  
134      * Sets proliferative type as stem cell  
135      ***/  
136  
137     void Initialise();  
138  
139     /**  
140      * Overridden InitialiseDaughterCell() method.  
141      * Used to implement asymmetric mitotic mode  
142      * Daughter cells are initialised w/ He cell cycle model  
143      * */  
144     void InitialiseDaughterCell();  
145  
146     /*Model setup functions for standard He (SetModelParameters) and  
147      → deterministic alternative (SetDeterministicMode) models  
148      * Default parameters are from refits of He et al + deterministic  
149      ← alternatives  
150      * He 2012 params: mitoticModePhase2 = 8, mitoticModePhase3 = 15,  
151      ← p1PP = 1, p1PD = 0, p2PP = .2, p2PD = .4, p3PP = .2, p3PD = 0  
152      * gammaShift = 4, gammaShape = 2, gammaScale = 1, sisterShift = 1  
153      */
```

```
151 void SetModelParameters(double gammaShift = 4, double gammaShape = 2,
152   ↵ double gammaScale = 1, std::vector<double> heParamVector = { 8,
153   ↵ 15, 1, 0, .2, .4, .2, 0, 4, 2, 1, 1 });
152 void EnableExpandingStemPopulation(int basePopulation,
153   ↵ boost::shared_ptr<AbstractCellPopulation<2>> p_population);
153 void SetTimeDependentCycleDuration(double peakRateTime, double
154   ↵ increasingSlope, double decreasingSlope);

154
155 //Functions to enable per-cell mitotic mode logging for mode rate &
156   ↵ sequence sampling fixtures
156 //Uses singleton logfile
157 void EnableModeEventOutput(double eventStart, unsigned seed);

158
159 //More detailed debug output. Needs a ColumnDataWriter passed to it
160 //Only declare ColumnDataWriter directory, filename, etc; do not set
161   ↵ up otherwise
161 void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
162   ↵ debugWriter);

162
163 //Not used, but must be overwritten lest WanStemCellCycleModels be
164   ↵ abstract
164 double GetAverageTransitCellCycleTime();
165 double GetAverageStemCellCycleTime();

166 /**
167 * Overridden OutputCellCycleModelParameters() method.
168 *
169 * @param rParamsFile the file stream to which the parameters are
170   ↵ output
171 */
172 virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
173 };

174
175 #include "SerializationExportWrapper.hpp"
176 // Declare identifier for the serializer
177 CHASTE_CLASS_EXPORT(WanStemCellCycleModel)

178
179 #endif /*WANSTEMCELLCYCLEMODEL_HPP_*/
```

15.2 BioBackgroundModels

15.2.1 /README.md

```

1 ## BioBackgroundModels
2 [![Build
  ↳ Status](https://travis-ci.org/mmattocks/BioBackgroundModels.jl.svg?branch=master)
3 [![codecov](https://codecov.io/gh/mmattocks/BioBackgroundModels.jl/branch/master/graph/badge.svg)]
4 [![Project Status: WIP – Initial development is in progress, but there
  ↳ has not yet been a stable, usable release suitable for the
  ↳ public.](https://www.repostatus.org/badges/latest/wip.svg)](https://www.repostatus.org/badges/latest/wip.svg)

```

15.2.2 /src/BioBackgroundModels.jl

```

1 module BioBackgroundModels
2
3 import Base:copy,size
4 import Distances: euclidean
5 import Distributions:Univariate,Dirichlet,Categorical,logpdf,isprobvec
6 import Distributed: RemoteChannel, myid, remote_do, rmprocs
7 import HMMBase: AbstractHMM, assert_hmm, istransmat
8 import MCMCChains: Chains, ChainDataFrame, heideldiag
9 import Printf: @sprintf
10 import Random: rand, shuffle
11 import Serialization: serialize
12 import StatsFuns: logsumexp, logaddexp
13 import Statistics: mean
14 import UnicodePlots: lineplot,lineplot!, scatterplot,scatterplot!
15 using BioSequences, DataFrames, FASTX, GFF3, ProgressMeter
16
17 include("BHMM/BHMM.jl")
18 export BHMM
19 include("EM/chain.jl")
20 export Chain_ID, EM_step
21 include("API/genome_sampling.jl")
22 export setup_sample_jobs, execute_sample_jobs
23 include("API/EM_master.jl")
24 export setup_EM_jobs!, execute_EM_jobs!
25
26 include("reports/chain_report.jl")
27 include("reports/partition_report.jl")
28 include("reports/replicate_convergence.jl")

```

```

29 include("API/reports.jl")
30 export generate_reports
31
32 include("EM/baum-welch.jl")
33 include("EM/churbanov.jl")
34 include("utilities/load_balancer.jl")
35 export LoadConfig
36 include("EM/EM_converge.jl")
37 include("genome_sampling/partition_masker.jl")
38 include("genome_sampling/sequence_sampler.jl")
39 include("likelihood_funcs/bg_lh_matrix.jl")
40 export BGHMM_likelihood_calc
41 include("likelihood_funcs/hmm.jl")
42 export obs_lh_given_hmm
43 include("utilities/observation_coding.jl")
44 include("utilities/BBG_analysis.jl")
45 include("utilities/BBG_progressmeter.jl")
46 include("utilities/HMM_init.jl")
47 include("utilities/model_display.jl")
48 include("utilities/utilities.jl")
49 export split_obs_sets
50 include("utilities/log_prob_sum.jl")
51 export lps
52 end # module

```

15.2.3 /src/API/EM_master.jl

```

1 #function to setup an BHMM chains dictionary and RemoteChannel for
  ↳ learning jobs, given a vector of state #s, order_nos, replicates to
  ↳ train, the dictionary to fill, the RemoteChannel and the training
  ↳ sequences
2 #resumes any existing non-converged chains, otherwise initialises hmms
  ↳ for new chains given provided constants
3 function setup_EM_jobs!(job_ids::Vector{Chain_ID},
  ↳ obs_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}};
  ↳ delta_thresh::Float64=1e-3,
  ↳ chains::Dict{Chain_ID,Vector{EM_step}}=Dict{Chain_ID,Vector{EM_step}}(),
  ↳ init_function::Function=autotransition_init)
  ↳ #argument checking
  ↳ length(job_ids) < 1 && throw(ArgumentError("Empty job id vector!"))
  ↳ length(obs_sets) < 1 && throw(ArgumentError("Empty observation
    ↳ sets!"))

```

```

7
8     no_input_hmms = length(job_ids)
9     input_channel= RemoteChannel(()→Channel{Tuple}(no_input_hmms*3))
10    ↵  #channel to hold BHMM learning jobs
11    output_channel= RemoteChannel(()→Channel{Tuple}(Inf)) #channel to
12    ↵  take EM iterates off of
13
14    @showprogress 1 "Setting up HMMs ... " for id in job_ids #for each
15    ↵  jobid, add an initial BHMM to input_channel for EM_workers
16    if haskey(chains, id) && length(chains[id]) > 0 #true if resuming
17    ↵  from incomplete chain
18    chain_end=chains[id][end]
19    if !chain_end.converged || (chain_end.converged &&
20    ↵  chain_end.delta > delta_thresh)#push the last hmm iterate
21    ↵  for nonconverged chains to the input channel with coded
22    ↵  observations and values for chain resumption
23    ↵  put!(input_channel, (id, chain_end.iterate,
24    ↵      chain_end.hmm, chain_end.log_p, code_dict[(id.obs_id,
25    ↵          id.order)]))
26    else #skip any jobs that have converged from previous runs
27    ↵  no_input_hmms -= 1
28    end
29
30    else
31        hmm = init_function(id.K, id.order, id.obs_id) #initialise
32        ↵  first BHMM in chain
33        chains[id] = Vector{EM_step}() #initialise the relevant chain
34        obs=code_dict[(id.obs_id,id.order)]
35        lh=obs_lh_given_hmm(obs,hmm,linear=false)
36        put!(input_channel, (id, 1, hmm, lh, obs))
37    end
38
39    return no_input_hmms, chains, input_channel, output_channel
40 end
41
42 function execute_EM_jobs!(worker_pool::Vector{Int64},
43    ↵  no_input_hmms::Integer, chains::Dict{Chain_ID,Vector{EM_step}},
44    ↵  input_channel::RemoteChannel, output_channel::RemoteChannel,
45    ↵  chains_path::String; load_dict=Dict{Int64,LoadConfig}(),
46    ↵  EM_func::Function=linear_step, delta_thresh=1e-3, max_iterates=5000,
47    ↵  verbose=false)

```

```

35     #argument checking
36     length(worker_pool) < 1 && throw(ArgumentError("Worker pool must
37         ↳ contain one or more worker IDs!"))
38     no_input_hmms < 1 && throw(ArgumentError("Zero input HMMs reported,
39         ↳ likely continuing from chains already converged beyond default
40         ↳ delta_thresh for setup_EM_jobs"))
41     length(chains) < 1 && throw(ArgumentError("No chains supplied, likely
42         ↳ job set from setup_EM_jobs passed incorrectly"))
43     !isready(input_channel) && throw(ArgumentError("BHMM input channel
44         ↳ has no contents, likely job set from setup_EM_jobs already
45         ↳ executed"))

46     #SEND BHMM FIT JOBS TO WORKERS
47     if isready(input_channel) > 0
48         @info "Fitting HMMs.."
49         #WORKERS FIT HMMS
50         for worker in worker_pool
51             if worker in keys(load_dict)
52                 remote_do(EM_converge!, worker, input_channel,
53                     ↳ output_channel, no_input_hmms,
54                     ↳ load_config=load_dict[worker], EM_func=EM_func,
55                     ↳ delta_thresh=delta_thresh, max_iterates=max_iterates,
56                     ↳ verbose=verbose)
57             else
58                 remote_do(EM_converge!, worker, input_channel,
59                     ↳ output_channel, no_input_hmms, EM_func=EM_func,
60                     ↳ delta_thresh=delta_thresh, max_iterates=max_iterates,
61                     ↳ verbose=verbose)
62             end
63         end
64     else
65         @warn "No input HMMs (all already converged?), skipping
66             ↳ fitting.."
67     end

68     #GET LEARNT HMMS OFF REMOTECHANNEL, SERIALISE AT EVERY ITERATION,
69     ↳ UPDATE PROGRESS METERS
70     job_counter=no_input_hmms
71     learning_meters=Dict{Chain_ID, ProgressHMM}()
72     overall_meter=Progress(no_input_hmms,"Overall batch progress:")

73     while job_counter > 0
74         wait(output_channel)

```

```

63     workerid, jobid, iterate, hmm, log_p, delta, converged, steptime
64         ↵ = take!(output_channel)
65 #either update an existing ProgressHMM meter or create a new one
66         ↵ for the job
67 if haskey(learning_meters, jobid) && iterate > 2
68     update!(learning_meters[jobid], delta, steptime)
69 else
70     offset = workerid - 1
71 if iterate ≤ 2
72     learning_meters[jobid] = ProgressHMM(delta_thresh,
73         ↵ "$jobid on Wk $workerid:", offset, 2)
74     update!(learning_meters[jobid], delta, steptime)
75 else
76     learning_meters[jobid] = ProgressHMM(delta_thresh,
77         ↵ "$jobid on Wk $workerid:", offset, iterate)
78     update!(learning_meters[jobid], delta, steptime)
79 end
80
81 end
82 #push the hmm and related params to the results_dict
83 push!(chains[jobid], EM_step(iterate, hmm, log_p, delta,
84     ↵ converged))
85 #try to serialize the results; catch interrupts and other errors
86     ↵ to prevent corruption
87 try
88     serialize(chains_path, chains)
89 catch e
90     @warn "Serializing failed!"
91     println(e)
92 end
93
94 #decrement the job counter, update overall progress meter, and
95     ↵ save the current results dict on convergence or max iterate
96 if converged || iterate == max_iterates
97     job_counter -= 1
98     ProgressMeter.update!(overall_meter,
99         ↵ (no_input_hmms-job_counter))
100    if !isready(input_channel) #if there are no more jobs to be
101        ↵ learnt, retire the worker
102        workerid!=1 && rmprocs(workerid)
103    end
104 end
105 end
106
107 end

```

```

97     #count converged & unconverged jobs, report results
98     converged_counter = 0
99     unconverged_counter = 0
100    for (id, chain) in chains
101        chain[end].converged = true ? (converged_counter += 1) :
102            (unconverged_counter += 1)
103    end
104
105   @info "Background HMM batch learning task complete,
106       $converged_counter converged jobs, $unconverged_counter jobs
107       failed to converge in $max_iterates iterates since job start."
108 end

```

15.2.4 /src/API/genome_sampling.jl

```

1 #function to partition genome and set up Distributed RemoteChannels so
2     # partitions can be sampled simultaneously
3 function setup_sample_jobs(genome_path::String,
4     genome_index_path::String, gff3_path::String,
5     sample_set_length::Integer, sample_window_min::Integer,
6     sample_window_max::Integer, perigenic_pad::Integer;
7     deterministic::Bool=false)
8     #argument checking
9     !ispather(genome_path) && throw(ArgumentError("Bad genome path!"))
10    !ispather(genome_index_path) && throw(ArgumentError("Bad genome index
11        path!"))
12    !ispather(gff3_path) && throw(ArgumentError("Bad gff3 path!"))
13    sample_set_length < 1 && throw(ArgumentError("Sample set length must
14        be a positive integer!"))
15    sample_window_min < 1 || sample_window_max < 1 &&
16        throw(ArgumentError("Sample window minimum and maximum bounds
17        must be positive integers!"))
18    sample_window_min ≥ sample_window_max && throw(ArgumentError("Sample
19        window minimum size must be smaller than maximum size"))
20    perigenic_pad < 0 && throw(ArgumentError("Perigenic pad must be 0 or
21        positive!"))
22
23    coordinate_partitions = partition_genome_coordinates(gff3_path,
24        perigenic_pad)
25    sample_sets = DataFrame[]

```

```

14     input_sample_jobs =
15         → RemoteChannel(()→Channel{Tuple}(length(coordinate_partitions)))
16         → #channel to hold sampling jobs
17     completed_sample_jobs =
18         → RemoteChannel(()→Channel{Tuple}(length(coordinate_partitions)))
19         → #channel to hold completed sample dfs
20     for (partitionid, partition) in coordinate_partitions
21         add_metacoordinates!(partition)
22         put!(input_sample_jobs, (genome_path, genome_index_path,
23             → partition, partitionid, sample_set_length, sample_window_min,
24             → sample_window_max, deterministic))
25     end
26     progress_channel = RemoteChannel(()→Channel{Tuple}(20))
27     return (input_sample_jobs, completed_sample_jobs, progress_channel,
28             → sample_set_length)
29 end
30
31 function
32     execute_sample_jobs(channels::Tuple{RemoteChannel,RemoteChannel,RemoteChannel,Int}
33     → worker_pool::Vector{Int64}; partitions::Integer=3)
34     input_sample_channel, completed_sample_channel, progress_channel,
35     → sample_set_length = channels
36
37     #argument checking
38     length(worker_pool) < 1 && throw(ArgumentError("Worker pool must
39         → contain one or more worker IDs!"))
40     !isready(input_sample_channel) && throw(ArgumentError("Input sample
41         → channel not ready, likely channel set from setup_sample_jobs
42         → passed incorrectly"))
43     sample_set_length < 1 && throw(ArgumentError("Sample set length must
44         → be a positive integer, likely channel set from setup_sample_jobs
45         → passed incorrectly"))
46
47     #send sampling jobs to workers
48     if isready(input_sample_channel) > 0
49         @info "Sampling.."
50         #WORKERS SAMPLE
51         for (n, worker) in enumerate(worker_pool)
52             if n ≤ partitions #no more workers than partitions to be
53                 → used
54                 remote_do(get_sample_set, worker, input_sample_channel,
55                         → completed_sample_channel, progress_channel)
56             end
57         end
58     end
59 
```

```
40         end
41     else
42         @error "No sampling jobs!"
43     end
44
45     #progress meters for sampling
46     sampling_meters=Dict{String, Progress}()
47     overall_sampling_meter=Progress(partitions, "Overall sampling
48         → progress:")
48     completed_counter = 0
49     ProgressMeter.update!(overall_sampling_meter, completed_counter)
50     sampling_offset = ones(Bool, partitions)
51
52     #collect progress updates while waiting on completion of sampling
53         → jobs
54     while completed_counter < partitions
55         wait(progress_channel)
56         partition_id, progress = take!(progress_channel)
57         if haskey(sampling_meters, partition_id)
58             ProgressMeter.update!(sampling_meters[partition_id],
59                 → progress)
60         else
61             offset = findfirst(sampling_offset)[1]
62             sampling_meters[partition_id] = Progress(sample_set_length,
63                 → "Sampling partition $partition_id:", offset)
64             ProgressMeter.update!(sampling_meters[partition_id],
65                 → progress)
66             sampling_offset[offset] = false
67         end
68         if progress == sample_set_length
69             completed_counter += 1
70             ProgressMeter.update!(overall_sampling_meter,
71                 → completed_counter)
72         end
73     end
74
75     #collect sample dfs by partition id when ready
76     sample_record_dfs = Dict{String,DataFrame}()
77     collected_counter = 0
78     while collected_counter < partitions
79         wait(completed_sample_channel)
80         partition_id, sample_df = take!(completed_sample_channel)
81         sample_record_dfs[partition_id] = sample_df
```

```

77     collected_counter += 1
78     @info "Partition $partition_id completed sampling...""
79   end
80
81   return sample_record_dfs
82 end

```

15.2.5 /src/API/reports.jl

```

1 struct Report_Folder
2   partition_id::String
3   partition_report::Partition_Report
4   replicate_report::Replicate_Report
5   chain_reports::Dict{Chain_ID,Chain_Report}
6 end
7
8 function generate_reports(chains::Dict{Chain_ID,Vector{EM_step}},
9   → test_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}})
9   #check chains dict for problems
10
11  length(chains)==0 && throw(ArgumentError("Empty chains dict!"))
12  any(chain→length(chain)<2, values(chains)) &&
13    → throw(ArgumentError("Some chains are too short (<2 EM steps)!")
13    → "Probably not all chains have been operated on by EM workers yet.
13    → Try EM_converge first!"))
14  unconv=sum(!chain[end].converged for chain in values(chains))
14  unconv > 0 && @warn "Not all chains are converged to the selected
15    → step delta."
15  length(test_sets)==0 && throw(ArgumentError("Empty test_sets dict!"))
16
17  chain_reports = report_chains(chains, test_sets)
18  partition_reports = report_partitions(chain_reports)
19
20  report_folders=Dict{String, Report_Folder}()
21
22  for part_report in partition_reports
23    rep_report=report_replicates(part_report.best_repset, chains)
24    chain_subset=Dict{Chain_ID,Chain_Report}()
25    for id in keys(chain_reports)
26      id.obs_id=part_report.partition_id &&
26      → (chain_subset[id]=chain_reports[id])
27    end

```

```

28         ↵ report_folders[part_report.partition_id]=Report_Folder(part_report.partit
29             ↵ part_report, rep_report, chain_subset)
30     end
31
32     return report_folders
33 end

```

15.2.6 /src/BHMM/BHMM.jl

```

1 """
2     BHMM([a::AbstractVector{T}, ]A::AbstractMatrix{T},
3         ↵ B::AbstractVector{<:Categorical}) where F where T
4 Build an BHMM with transition matrix `A` and observations distributions
5         ↵ `B`.
6 If the initial state distribution `a` is not specified, a uniform
7         ↵ distribution is assumed.
8 Observations distributions can be of different types (for example
9         ↵ `Normal` and `Exponential`).
10 However they must be of the same dimension (all scalars or all
11         ↵ multivariates).
12 # Example
13 """
14 """
15 struct BHMM{T} <: AbstractHMM{Univariate}
16     a::AbstractVector{T}
17     A::AbstractMatrix{T}
18     B::AbstractVector{<:Categorical}
19     partition::String
20     BHMM{T}(a, A, B, partition="") where {T} = assert_BHMM(a, A, B) &&
21         ↵ new(a, A, B, partition)
22 end
23
24 BHMM(a::AbstractVector{T}, A::AbstractMatrix{T},
25         ↵ B::AbstractVector{<:Categorical}, partition ...) where {T} =
26         ↵ BHMM{T}(a, A, B, partition ...)
27 BHMM(A::AbstractMatrix{T}, B::AbstractVector{<:Categorical},
28         ↵ partition ...) where {T} = BHMM{T}(ones(size(A)[1])/size(A)[1], A, B,
29         ↵ partition ...)
30
31 """
32 """

```

```

23 copy(hmm::BHMM) = BHMM(copy(hmm.a), copy(hmm.A), copy(hmm.B),
24   ↪ hmm.partition)
25 size(hmm::BHMM) = (length(hmm.B), length(hmm.B[1].p))
26
27 """
28     assert_BHMM(a, A, B)
29 Throw an `ArgumentError` if the initial state distribution and the
30   ↪ transition matrix rows does not sum to 1,
31 and if the observations distributions does not have the same dimensions.
32 """
33 function assert_BHMM(a::AbstractVector,
34                       A::AbstractMatrix,
35                       B::AbstractVector{<:Categorical})
36   !isprobvec(a) && throw(ArgumentError("Initial state vector a is not a
37   ↪ valid probability vector!"))
38   !istransmat(A) && throw(ArgumentError("Transition matrix A is not
39   ↪ valid!"))
40
41   !all([length(d.p) for d in B] .== length(B[1].p)) &&
42   ↪ throw(ArgumentError("All distributions must have the same
43   ↪ dimensions"))
44   !(length(a) == size(A,1) == length(B)) && throw(ArgumentError("Length
45   ↪ of initial state vector a, dimension of transition matrix A, and
46   ↪ number of distributions B are not the same!"))
47   return true
48 end
49
50 function Base.show(io::IO, hmm::BHMM)
51   println(io, "Background BHMM")
52   println(io, "State Initial and Transition Probabilities")
53   print(io, "a: ")
54   show(io, hmm.a)
55   println(io)
56   print(io, "A: ")
57   display(hmm.A)
58   println(io)
59   println(io, "INFORMATIVE SYMBOLS BY STATE")
60   for (n,d) in enumerate(hmm.B)
61     print(io, "K$n ")
62     print_emitters(d)
63   end
64 end
65
66 end

```

15.2.7 /src/EM/EM_converge.jl

```

1 function EM_converge!(hmm_jobs :: RemoteChannel,
2   ↳ output_hmms :: RemoteChannel, no_models :: Integer;
3   ↳ load_config :: LoadConfig=LoadConfig(1:typemax(Int64)-1,
4   ↳ 0:typemax(Int64)-1), EM_func :: Function=linear_step,
5   ↳ delta_thresh=1e-3, max_iterates=5000, verbose=false)
6   ↳ while isready(hmm_jobs)
7     ↳   workerid = myid()
8     ↳   jobid, start_iterate, hmm, job_norm, observations =
9     ↳     ↳   load_balancer(no_models, hmm_jobs, load_config)
10    ↳   jobid = 0 && break #no valid job for this worker according to
11    ↳     ↳   load_table entry
12
13    ↳   start_iterate > max_iterates - 1 && throw(ArgumentError("BHMM
14    ↳     ↳   chain $jobid is already longer ($start_iterate iterates) than
15    ↳     ↳   specified max_iterates!"))
16    ↳   curr_iterate = start_iterate
17
18    ↳   #mask calculations here rather than ms_mle_step to prevent
19    ↳     ↳   recalculations every iterate
20    ↳   #build array of observation lengths
21
22    ↳   obs_lengths = [findfirst(iszero, observations[o,:])-1 for o in
23    ↳     ↳   1:size(observations,1)] #mask calculations here rather than
24    ↳     ↳   mle_step to prevent recalculations every iterate
25    ↳   EM_func=bw_step && (observations=transpose(observations))
26
27    ↳   start_iterate = 1 && put!(output_hmms, (workerid, jobid,
28    ↳     ↳   curr_iterate, hmm, job_norm, 0.0, false, 0.0)); #on the first
29    ↳     ↳   iterate return the initial BHMM immediately
30    ↳   verbose && @info "Fitting BHMM on Wk $workerid, start iterate
31    ↳     ↳   $start_iterate, $jobid with $(size(hmm)[1]) states and
32    ↳     ↳   $(size(hmm)[2]) symbols ... "
33
34    ↳   curr_iterate += 1
35
36    ↳   start=time()
37    ↳   new_hmm, last_norm = EM_func(hmm, observations, obs_lengths)
38    ↳   delta = abs(lps(job_norm, -last_norm))

```

```

24     put!(output_hmms, (workerid, jobid, curr_iterate, new_hmm,
25                           ↳ last_norm, delta, false, time()-start))
26
27     for i in curr_iterate:max_iterates
28         start=time()
29         new_hmm, norm = EM_func(new_hmm, observations, obs_lengths)
30         curr_iterate += 1
31         delta = abs(lps(norm, -last_norm))
32         if delta < delta_thresh
33             put!(output_hmms, (workerid, jobid, curr_iterate,
34                               ↳ new_hmm, norm, delta, true, time()-start))
35             verbose && @info "$jobid converged after"
36             ↳ $(curr_iterate-1) " EM steps"
37             break
38         else
39             put!(output_hmms, (workerid, jobid, curr_iterate,
40                               ↳ new_hmm, norm, delta, false, time()-start))
41             verbose && @info "$jobid EM step $(curr_iterate-1) delta"
42             ↳ $delta"
43             last_norm = norm
44         end
45     end
46   end
47 end

```

15.2.8 /src/EM/baum-welch.jl

```

1 """
2     ms_mle_step(hmm::AbstractHMM{F}, observations) where F
3
4 Perform one step of the EM (Baum-Welch) algorithm.
5
6 # Example
7 """
8 hmm, log_likelihood = ms_mle_step(hmm, observations)
9 """
10 """
11
12 function bw_step(hmm::BHMM, observations, obs_lengths)
13     lls = bw_llhs(hmm, observations)
14     log_α = messages_forwards_log(hmm.a, hmm.A, lls, obs_lengths)
15     log_β = messages_backwards_log(hmm.A, lls, obs_lengths)

```

```

16 log_A = log.(hmm.A)
17
18 K,Tmaxplus1,O = size(lls) #the last T value is the 0 end marker of
   ↵ the longest T
19
20 #transforms to cut down log_ξ, log_γ assignment times
21 lls = permutedims(lls, [2,3,1]) # from (K,T,O) to (T,O,K)
22 log_α = permutedims(log_α, [2,3,1])
23 log_β = permutedims(log_β, [2,3,1])
24
25 # E-step
26 log_ξ = fill(-Inf, Tmaxplus1,O,K,K)
27 log_γ = fill(-Inf, Tmaxplus1,O,K)
28 log_pobs = zeros(O)
29
30 @inbounds for o = 1:O
31     log_pobs[o] = logsumexp(lps.(log_α[1,o,:], log_β[1,o,:]))
32 end
33
34 Threads.@threads for idx in [(i,j,o) for i=1:K, j=1:K, o=1:O]
35     i,j,o = idx[1],idx[2],idx[3]
36     obsl = obs_lengths[o]
37     @inbounds for t = 1:obsl-1 #log_ξ & log_γ calculated to T-1 for
       ↵ each o
38         log_ξ[t,o,i,j] = lps(log_α[t,o,i], log_A[i,j], log_β[t+1,o,j],
           ↵ lls[t+1,o,j], -log_pobs[o])
39         log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
40     end
41     t=obsl #log_ξ @ T = 0
42     log_ξ[t,o,i,j] = 0
43     log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
44 end
45
46 ξ = similar(log_ξ)
47 ξ .= exp.(log_ξ)
48 Σk_ξ = sum(ξ, dims=[3,4])
49 nan_mask = Σk_ξ .== 0
50 Σk_ξ[nan_mask] .= Inf #prevent NaNs in dummy renorm
51 ξ ./= Σk_ξ #dummy renorm across K to keep numerical creep from
   ↵ causing isprobvec to fail on new new_A during hmm creation
52
53 γ = similar(log_γ)
54 γ .= exp.(log_γ)

```

```

55     Σk_γ = sum(y, dims=3)
56     Σk_γ[nan_mask[:, :, :]] .= Inf #prevent NaNs in dummy renorm
57     y .= Σk_γ #dummy renorm
58
59     # M-step
60     new_A = zeros(K,K)
61     for i=1:K, j=1:K
62         Σotξ_vec = zeros(0)
63         Σotγ_vec = zeros(0)
64         Threads.@threads for o in 1:O
65             Σotξ_vec[o] = sum(ξ[1:obs_lengths[o]-1,o,i,j])
66             Σotγ_vec[o] = sum(y[1:obs_lengths[o]-1,o,i])
67         end
68         new_A[i,j] = sum(Σotξ_vec) / sum(Σotγ_vec)
69     end
70     new_A .= sum(new_A, dims=2) #dummy renorm
71     new_a = (sum(y[1,:,:], dims=1)./sum(sum(y[1,:,:], dims=1)))[1,:]
72     new_a .= sum(new_a) #dummy renorm
73
74     obs_mask = .!nan_mask
75     obs_collection = observations[obs_mask[:, :]]
76
77     B = Categorical[]
78     @inbounds for (i, d) in enumerate(hmm.B)
79         γ_d = y[:, :, i]
80         push!(B, t_categorical_mle(Categorical, d.support[end],
81             → obs_collection, γ_d[obs_mask[:, :]]))
82     end
83
84     return typeof(hmm)(new_a, new_A, B), lps(log_pobs)
85 end
86
87     function bw_llhs(hmm, observations)
88         lls = zeros(length(hmm.B), size(observations)... )
89         Threads.@threads for d in 1:length(hmm.B)
90             lls[d,:,:] = logpdf.(hmm.B[d], observations)
91         end
92         return lls
93     end
94     #Multisequence competent log implementations of forward
95     → and backwards algos
96     function messages_forwards_log(init_distn, trans_matrix,
97         → log_likelihoods, obs_lengths)
98         log_alphas = zeros(size(log_likelihoods))

```

```

95         log_trans_matrix = log.(trans_matrix)
96         log_alphas[:,1,:] = log.(init_distn) .+
97             ↪ log_likelihoods[:,1,:]
98         Threads.@threads for o in 1:size(log_likelihoods)[3]
99             @inbounds for t in 2:obs_lengths[o], j in
100                 ↪ 1:size(log_likelihoods)[1]
101
102                 ↪ log_alphas[j,t,o]=logsumexp(log_alphas[:,t-1,o]
103                 ↪ .+ log_trans_matrix[:,j]) +
104                 ↪ log_likelihoods[j,t,o]
105             end
106         end
107         return log_alphas
108     end
109
110     function messages_backwards_log(trans_matrix,
111         ↪ log_likelihoods, obs_lengths)
112         log_betas = zeros(size(log_likelihoods))
113         log_trans_matrix = log.(trans_matrix)
114         Threads.@threads for o in 1:size(log_likelihoods)[3]
115             @inbounds for t in obs_lengths[o]-1:-1:1
116                 tmp = view(log_betas, :, t+1, o) .+
117                     ↪ view(log_likelihoods, :, t+1, o)
118                 @inbounds for i in 1:size(log_likelihoods)[1]
119                     log_betas[i,t,o] =
120                         ↪ logsumexp(view(log_trans_matrix, i,:)
121                         ↪ .+ tmp)
122                 end
123             end
124         end
125         return log_betas
126     end
127     #subfunc derived from Distributions.jl categorical.jl
128     ↪ fit_mle, threaded
129     function t_categorical_mle(::Type{<:Categorical},
130         ↪ k::Integer, x::AbstractArray{T}, w::AbstractArray{F})
131         ↪ where T<:Integer where F<:AbstractFloat
132
133             ↪ Categorical(t_pnormalize!(t_add_categorical_counts!(zeros(k),
134             ↪ x, w)))
135     end
136
137     t_pnormalize!(v::Vector) = (v ./= sum(v); v)

```

```

124
125     function t_add_categorical_counts!(h::Vector{F},
126         ↳ x::AbstractArray{T}, w::AbstractArray{F}) where
127         ↳ T<:Integer where F<:AbstractFloat
128             n = length(x)
129             if n ≠ length(w)
130                 throw(DimensionMismatch("Inconsistent array
131                 ↳ lengths."))
132             end
133             hlock = ReentrantLock()
134             Threads.@threads for i=1:n
135                 @inbounds xi = x[i]
136                 @inbounds wi = w[i]
137                 lock(hlock)
138                 h[xi] += wi    # cannot use @inbounds, as no
139                 ↳ guarantee that x[i] is in bound
140                 unlock(hlock)
141             end
142             return h
143         end

```

15.2.9 /src/EM/chain.jl

```

1 struct Chain_ID
2     obs_id::AbstractString
3     K::Integer
4     order::Integer
5     replicate::Integer
6     Chain_ID(obs_id,K,order,replicate) =
7         ↳ assert_chain_id(K,order,replicate) &&
8         ↳ new(obs_id,K,order,replicate)
9 end
10
11 function assert_chain_id(K, order, replicate)
12     K < 1 && throw(ArgumentError("Chain_ID K (# of hmm states) must be a
13     ↳ positive integer!"))
14     order < 0 && throw(ArgumentError("Chain_ID symbol order must be zero
15     ↳ or a positive integer!"))
16     replicate < 1 && throw(ArgumentError("Chain_ID replicate must be a
17     ↳ positive integer!"))
18     return true
19 end

```

```

15
16 struct EM_step
17     iterate::Integer
18     hmm::BHMM
19     log_p::AbstractFloat
20     delta::AbstractFloat
21     converged::Bool
22
23     → EM_step(iterate,hmm,log_p,delta,converged)=assert_step(iterate,hmm,log_p)
24     → && new(iterate, hmm, log_p, delta, converged)
25 end
26
27 function assert_step(iterate, hmm, log_p)
28     iterate < 1 && throw(ArgumentError("EM_step iterate number must be a
29     → positive integer!"))
30     assert_hmm(hmm.a, hmm.A, hmm.B)
31     log_p > 0.0 && throw(ArgumentError("EM_step log probability value
32     → must be 0 or negative!"))
33     return true
34 end

```

15.2.10 /src/EM/churbanov.jl

```

1 function linear_step(hmm, observations, obs_lengths)
2     O,T = size(observations);
3     a = transpose(log.(hmm.a)); A = log.(hmm.A) #less expensive to
4     → transpose transmatrix in the backwards_sweep than to transpose
5     → here and take a ranged view ie view(A,m:m,:) is more expensive
6     → than transpose(view(A,m,:))
7     N = length(hmm.B); Γ = length(hmm.B[1].support);
8     mask=observations.≠0
9     #INITIALIZATION
10    βoi_T = zeros(O,N); βoi_t = zeros(O,N) #log betas at T initialised as
11    → zeros
12    Eoyim_T = fill(-Inf,O,Γ,N,N); Eoyim_t = fill(-Inf,O,Γ,N,N)
13    for m in 1:N, i in 1:N, y in 1:Γ, o in 1:O
14        observations[o, obs_lengths[o]] = y && m == i && (Eoyim_T[o, y,
15        → i, m] = 0)
16    end
17    Tijm_T = fill(-Inf,O,N,N,N); Tijm_t = fill(-Inf,O,N,N,N) #Ti,j(T,m) =
18    → 0 for all m; in logspace
19

```

```

14 #RECURRENCE
15
16     → βoi_T,Tijm_T,Eoyim_T=backwards_sweep!(hmm,A,N,Γ,βoi_T,βoi_t,Tijm_T,Tijm_t,Eoy
17
18 #TERMINATION
19 lls = c_llhs(hmm,observations[:,1])
20 α1om = lps.(lls,a) #first position forward msgs
21 log_pobs = [c_lse(lps.(α1om[o,:], βoi_T[o,:])) for o in 1:0]
22
23 Toij = [c_lse([lps(Tijm_T[o,i,j,m], α1om[o,m], -log_pobs[o]) for m in
24     → 1:N]) for o in 1:0, i in 1:N, j in 1:N] #terminate Tij with
25     → forward messages
26 Eoiy = [c_lse([lps(Eoyim_T[o,y,i,m], α1om[o,m], -log_pobs[o]) for m
27     → in 1:N]) for o in 1:0, i in 1:N, y in 1:Γ] #terminate Eids with
28     → forward messages
29
30 #INTEGRATE ACROSS OBSERVATIONS AND SOLVE FOR NEW BHMM PARAMS
31 #INITIAL STATE DIST
32 a_o=α1om.+βoi_T.-c_lse.(eachrow(α1om.+βoi_T)) #estimate a for each o
33 obs_penalty=log(0) #broadcast subtraction to normalise log prob vals
34     → by obs number
35 new_a=c_lse.(eachcol(a_o))-obs_penalty #sum over obs and normalise
36     → by number of obs
37 #TRANSITION MATRIX
38 new_A = [c_lse(view(Toij,:,:i,j)) for i in 1:N, j in
39     → 1:N]-c_lse.(eachcol(c_lse.([view(Toij,o,i,:) for o in 1:0, i in
40     → 1:N])))
41 #EMISSION MATRIX
42 new_b=[c_lse(view(Eoiy,:,:i,y)) for i in 1:N, y in
43     → 1:Γ]-c_lse.(eachcol(c_lse.([view(Eoiy,o,i,:) for o in 1:0, i in
44     → 1:N])))
45 new_B=[Categorical(exp.(new_b[i,:])) for i in 1:N]
46
47 return BHMM(exp.(new_a), exp.(new_A), new_B, hmm.partition),
48     → lps(log_pobs)
49 end
50
51 #LINEAR_STEP SUBFUNCS
52 function backwards_sweep!(hmm, A, N, Γ, βoi_T, βoi_t,
53     → Tijm_T, Tijm_t, Eoyim_T, Eoyim_t, observations, mask,
54     → obs_lengths)
55     for t in maximum(obs_lengths)-1:-1:1
56         last_β=copy(βoi_T)
57         lls = c_llhs(hmm,observations[:,t+1])
58         α1om = lps.(lls,a) #first position forward msgs
59         log_pobs = [c_lse(lps.(α1om[o,:], βoi_T[o,:])) for o in 1:0]
60         Toij = [c_lse([lps(Tijm_T[o,i,j,m], α1om[o,m], -log_pobs[o]) for m in
61             → 1:N]) for o in 1:0, i in 1:N, j in 1:N] #terminate Tij with
62             → forward messages
63         Eoiy = [c_lse([lps(Eoyim_T[o,y,i,m], α1om[o,m], -log_pobs[o]) for m
64             → in 1:N]) for o in 1:0, i in 1:N, y in 1:Γ] #terminate Eids with
65             → forward messages
66
67         new_a=c_lse.(eachcol(a_o))-obs_penalty #sum over obs and normalise
68             → by number of obs
69         new_A = [c_lse(view(Toij,:,:i,j)) for i in 1:N, j in
70             → 1:N]-c_lse.(eachcol(c_lse.([view(Toij,o,i,:) for o in 1:0, i in
71                 → 1:N])))
72         new_b=[c_lse(view(Eoiy,:,:i,y)) for i in 1:N, y in
73             → 1:Γ]-c_lse.(eachcol(c_lse.([view(Eoiy,o,i,:) for o in 1:0, i in
74                 → 1:N])))
75         new_B=[Categorical(exp.(new_b[i,:])) for i in 1:N]
76
77         return BHMM(exp.(new_a), exp.(new_A), new_B, hmm.partition),
78             → lps(log_pobs)
79     end
80 end

```

```

43         omask = findall(mask[:,t+1])
44         βoi_T[omask,:] .+= view(lls,omask,:)
45         Threads.@threads for m in 1:N
46             βoi_t[omask,m] =
47                 ↳ c_lse.(eachrow(view(βoi_T,omask,:).+transpose(view(A,
48                 ↳ for j in 1:N, i in 1:N
49                     Tijm_t[omask, i, j, m] :=
50                         ↳ c_lse.(eachrow(lps.(view(Tijm_T,omask,i,j,:),
51                             ↳ view(lls,omask,:),
52                             ↳ transpose(view(A,m,:))))))
53                         i==m && (Tijm_t[omask, i, j, m] :=
54                             ↳ logaddexp.(Tijm_t[omask, i, j, m],
55                             ↳ lps.(last_β[omask,j],
56                             ↳ A[m,j],lls[omask,j])))
57             end
58             for i in 1:N, y in 1:T
59                 Eoyim_t[omask, y, i, m] :=
60                     ↳ c_lse.(eachrow(lps.(view(Eoyim_T,omask,y,i,:)),view(
61                         ↳ if i==m
62                             symmask =
63                                 ↳ findall(observations[:,t]==y)
64                             Eoyim_t[symmask, y, i, m] :=
65                                 ↳ logaddexp.(Eoyim_t[symmask, y, i,
66                                 ↳ m], βoi_t[symmask,m])
67                         end
68                         end
69                     end
70                     βoi_T=copy(βoi_t); Tijm_T=copy(Tijm_t); Eoyim_T =
71                         ↳ copy(Eoyim_t);
72                 end
73                 return βoi_T, Tijm_T, Eoyim_T
74             end
75
76             #logsumexp
77             function c_lse(X::AbstractArray{T}; dims=:) where
78                 ↳ {T<:Real}
79                 u=maximum(X)
80                 u isa AbstractArray || isfinite(u) || return float(u)
81                 return u + log(sum(x → exp(x-u), X))
82             end
83
84             #log likelihoods
85             function c_llhs(hmm, observation)

```

```

73         lls = zeros(length(observation),length(hmm.B))
74         Threads.@threads for d in 1:length(hmm.B)
75             lls[:,d] = logpdf.(hmm.B[d], observation)
76         end
77         return lls
78     end

```

15.2.11 /src/genome_sampling/partition_masker.jl

```

1 function get_partition_code_dict(dict_forward::Bool=true)
2     if dict_forward == true
3         return partition_code_dict =
4             Dict("intergenic"⇒1,"periexonic"⇒2,"exon"⇒3)
5     else
6         return code_partition_dict = Dict(1⇒"intergenic",
7             2⇒"periexonic", 3⇒"exon")
8     end
9 end
10
11 function make_padded_df(position_fasta::String, gff3_path::String,
12     → genome_path::String, genome_index_path::String, pad::Integer)
13     position_reader = FASTA.Reader(open((position_fasta), "r"))
14     genome_reader = open(FASTA.Reader, genome_path,
15         → index=genome_index_path)
16     scaffold_df = build_scaffold_df(gff3_path)
17     position_df = DataFrame(SeqID = String[], Start=Int[], End=Int[],
18         → PadSeq = LongSequence[], PadStart=Int[], RelStart=Int[],
19         → SeqOffset=Int[])
20     scaffold_seq_dict = build_scaffold_seq_dict(genome_path,
21         → genome_index_path)
22
23     for entry in position_reader
24         scaffold = FASTA.identifier(entry)
25
26         if scaffold ≠ "MT"
27             desc_array = split(FASTA.description(entry))
28             pos_start = parse(Int, desc_array[2])
29             pos_end = parse(Int, desc_array[4])
30             scaffold_end = scaffold_df.End[findfirst(isequal(scaffold),
31                 → scaffold_df.SeqID)]
32
33             pad_start=max(1, pos_start-pad)
34
35         end
36     end
37
38     return position_df, scaffold_df, scaffold_seq_dict
39 end

```

```

26         pad_length= pos_start - pad_start
27         seq_offset = pad - pad_length
28         padded_seq = fetch_sequence(scaffold, scaffold_seq_dict,
29             ↪ pad_start, pos_end, '+')
30
31     if !hasambiguity(padded_seq)
32         push!(position_df, [scaffold, pos_start, pos_end,
33             ↪ padded_seq, pad_start, pad_length, seq_offset])
34     end
35   end
36
37   close(position_reader)
38   close(genome_reader)
39   return position_df
40 end
41
42 function add_partition_masks!(position_df::DataFrame, gff3_path::String,
43   ↪ perigenic_pad::Integer=500,
44   ↪ columns::Tuple{Symbol,Symbol,Symbol}=(:SeqID, :PadSeq, :PadStart))
45   partitions=["exon", "periexonic", "intergenic"]
46   partition_coords_dict = partition_genome_coordinates(gff3_path,
47     ↪ perigenic_pad)
48   partitioned_scaffolds =
49     ↪ divide_partitions_by_scaffold(partition_coords_dict)
50   maskcol = [zeros(Int64,0,0) for i in 1:size(position_df,1)]
51   position_df.MaskMatrix=maskcol
52
53   @Threads.threads for entry in eachrow(position_df)
54     scaffold = entry[columns[1]]
55     maskLength = length(entry[columns[2]])
56     seqStart = entry[columns[3]]
57
58     scaffold_coords_dict = Dict{String,DataFrame}()
59
60     for ((partition, part_scaffold), df) in partitioned_scaffolds
61       if scaffold == part_scaffold
62         scaffold_coords_dict[partition] = df
63       end
64     end
65
66     entry.MaskMatrix=mask_sequence_by_partition(maskLength, seqStart,
67       ↪ scaffold_coords_dict)

```

```

62     end
63
64 end
65 #add_partition_masks!() SUBFUNCTIONS
66 function
67     → divide_partitions_by_scaffold(partition_coords_dict::Dict{String,
68     → DataFrame})
69     scaffold_coords_dict = Dict{Tuple{String, String}, DataFrame}()
70     for (partition_id, partition_df) in
71     partition_coords_dict
72         for scaffold_subframe in groupby(partition_df,
73             → :SeqID)
74             scaffold_id = scaffold_subframe.SeqID[1]
75             scaffold_df = copy(scaffold_subframe)
76             scaffold_coords_dict[(partition_id,
77             → scaffold_id)] = scaffold_df
78         end
79     end
80     return scaffold_coords_dict
81 end
82
83
84 function mask_sequence_by_partition(maskLength::Integer,
85     → seqStart::Integer, scaffold_coords_dict::Dict{String,
86     → DataFrame})
87     partition_code_dict = get_partition_code_dict()
88     seqMask = zeros(Integer, (maskLength, 2))
89     position = seqStart
90     while position ≤ seqStart+maskLength
91         position_partition, partition_extent,
92         → position_strand =
93         → find_position_partition(position,
94         → scaffold_coords_dict)
95
96         partition_code =
97         → partition_code_dict[position_partition]
98         mask_position = position - seqStart + 1
99
100        → seqMask[mask_position:min(maskLength,mask_position
101        → + partition_extent),1] .= partition_code
102        if position_strand == '+'
103    end
104
105    return seqMask
106 end

```

```

89
90         ↵ seqMask[mask_position:min(maskLength,mask_position
91         ↵ + partition_extent),2] *= 1
92     elseif position_strand == '-'
93
94         ↵ seqMask[mask_position:min(maskLength,mask_position
95         ↵ + partition_extent),2] *= -1
96     else
97
98         ↵ seqMask[mask_position:min(maskLength,mask_position
99         ↵ + partition_extent),2] *= 0
100    end
101
102    position += partition_extent + 1
103  end
104
105  return seqMask
106 end
107
108 function find_position_partition(position::Integer,
109     ↵ partition_dict::Dict{String, DataFrame})
110     foundPos = false
111     position_partition_id = ""
112     three_prime_extent = 0
113     sample_strand = 0
114     for (partition_id, partition) in partition_dict
115         hitindex = findfirst(x→x≥position,
116             ↵ partition.End)
117         if hitindex ≠ nothing
118             if position ≥ partition.Start[hitindex]
119                 foundPos=true
120                 position_partition_id = partition_id
121                 three_prime_extent =
122                     ↵ partition.End[hitindex] - position
123                 if partition_id == "exon" || partition_id
124                     ↵ == "perixonic"
125                     sample_strand =
126                         ↵ partition.Strand[hitindex]
127                 end
128             end
129         end
130     end
131 end

```

```

121             if foundPos == false
122                 throw(DomainError("Position $position not found
123                               ↪ among partition coordinates!"))
124             else
125                 return position_partition_id, three_prime_extent,
126                               ↪ sample_strand
127             end
128         end
129
130 #function to partition a genome into coordinate sets of:
131 #merged exons
132 #"periexonic" sequences (genes with 5' and 3' boundaries projected
133   ↪ -/+perigenic_pad bp, minus exons) - includes promoter elements,
134   ↪ introns, 3' elements
135 #intergenic sequences (everything else)
136 #given a valid gff3
137 function partition_genome_coordinates(gff3_path::String,
138   ↪ perigenic_pad::Integer=500)
139   # construct dataframes of scaffolds and metacoordinates
140   scaffold_df = build_scaffold_df(gff3_path)
141
142   #partition genome into intragenic, periexonic, and exonic coordinate
143   ↪ sets
144   #assemble exonic featureset
145   exon_df = DataFrame(SeqID = String[], Start = Integer[], End =
146     ↪ Integer[], Strand=Char[])
147   build_feature_df!(gff3_path, "CDS", "MT", exon_df)
148
149   #project exon coordinates onto the scaffold bitwise, merging
150   ↪ overlapping features and returning the merged dataframe
151   merged_exon_df = DataFrame(SeqID = String[], Start = Integer[], End =
152     ↪ Integer[], Strand=Char[])
153   @showprogress 1 "Partitioning exons ..." for scaffold_subframe in
154     ↪ DataFrames.groupby(exon_df, :SeqID) # for each scaffold subframe
155     ↪ that has exon features
156     scaffold_id = scaffold_subframe.SeqID[1] #get the scaffold id
157     scaffold_bitarray = init_scaffold_bitarray(scaffold_id,
158       ↪ scaffold_df, false, stranded=true) #init a stranded bitarray
159       ↪ of scaffold length
160
161     ↪ project_features_to_bitarray!(scaffold_subframe, scaffold_bitarray)
162     ↪ #project features as Trues on bitarray of falses

```

```

148     merged_subframe = get_feature_df_from_bitarray(scaffold_id,
149         ← scaffold_bitarray) #get a feature df from the projected
150         ← bitarray
151     append!(merged_exon_df,merged_subframe) #append the merged
152         ← scaffold df to the overall merged df
153 end
154
155 #assemble gene featureset
156 gene_df = DataFrame(SeqID = String[], Start = Integer[], End =
157     ← Integer[], Strand = Char[])
158 build_feature_df!(gff3_path, "gene", "MT", gene_df)
159
160 perigenic_pad > 0 && add_pad_to_coordinates!(gene_df, scaffold_df,
161     ← perigenic_pad) #if a perigenic pad is specified (to capture
162     ← promoter/downstream elements etc in the periexonic set), apply it
163     ← to the gene coords
164
165 #build intergenic coordinate set by subtracting gene features from
166     ← scaffold bitarrays
167 intergenic_df = DataFrame(SeqID = String[], Start = Integer[], End =
168     ← Integer[])
169 @showprogress 1 "Partitioning intergenic regions ..." for
170     ← scaffold_subframe in DataFrames.groupby(scaffold_df, :SeqID) #
171     ← for each scaffold subframe that has exon features
172     scaffold_id = scaffold_subframe.SeqID[1] #get the scaffold id
173     scaffold_bitarray = init_scaffold_bitarray(scaffold_id,
174         ← scaffold_df, true) #init a bitarray of scaffold length
175     if any(isequal(scaffold_id),gene_df.SeqID) #if any genes on the
176         ← scaffold
177         scaffold_genes=gene_df[findall(isequal(scaffold_id),
178             ← gene_df.SeqID), :] #get the gene rows by finding the
179             ← scaffold_id
180
181             ← subtract_features_from_bitarray!(scaffold_genes,scaffold_bitarray)
182             ← #subtract the gene positions from the scaffold bitarray
183     end
184     intragenic_subframe = get_feature_df_from_bitarray(scaffold_id,
185         ← scaffold_bitarray) #get a feature df from the projected
186         ← bitarray
187     append!(intergenic_df,intragenic_subframe) #append the merged
188         ← scaffold df to the overall merged df
189 end
190
191

```

```

171     #build periexonic set by projecting gene coordinates onto the
172     → scaffold bitwise, then subtracting the exons
173     periexonic_df = DataFrame(SeqID = String[], Start = Integer[], End =
174     → Integer[], Strand=Char[])
175     @showprogress 1 "Partitioning periexonic regions ... " for
176     → gene_subframe in DataFrames.groupby(gene_df, :SeqID) # for each
177     → scaffold subframe that has exon features
178     scaffold_id = gene_subframe.SeqID[1] #get the scaffold id
179     scaffold_bitarray = init_scaffold_bitarray(scaffold_id,
180     → scaffold_df, false, stranded=true) #init a bitarray of
181     → scaffold length
182     project_features_to_bitarray!(gene_subframe,scaffold_bitarray)
183     → #project features as Trues on bitarray of falses
184     if any(isequal(scaffold_id),merged_exon_df.SeqID)
185         scaffold_exons=merged_exon_df[findall(isequal(scaffold_id),
186         → merged_exon_df.SeqID), :]
187
188         → subtract_features_from_bitarray!(scaffold_exons,scaffold_bitarray)
189     end
190     periexonic_subframe = get_feature_df_from_bitarray(scaffold_id,
191     → scaffold_bitarray) #get a feature df from the projected
192     → bitarray
193     append!(periexonic_df,periexonic_subframe) #append the merged
194     → scaffold df to the overall merged df
195 end
196
197
198     #partition_genome_coordinates() SUBFUNCTIONS
199     #BITARRAY SCAFFOLD REPRESENTATION SUBFUNCTIONS
200     function init_scaffold_bitarray(scaffold_id::String,
201     → scaffold_df, value::Bool; stranded::Bool=false)
202     scaffold_length =
203     → scaffold_df.End[findfirst(isequal(scaffold_id),
204     → scaffold_df.SeqID)]
205     if stranded
206         value ? (return rscaffold_bitarray =
207         → trues(scaffold_length,2)) : (return
208         → rscaffold_bitarray = falses(scaffold_length,2))
209     else

```

```

195             value ? (return rscaffold_bitarray =
196             ↵   trues(scaffold_length,1)) : (return
197             ↵   rscaffold_bitarray = falses(scaffold_length,1))
198         end
199     end
200
201     function
202         → project_features_to_bitarray!(scaffold_feature_sf :: SubDataFrame,
203         → scaffold_bitarray :: BitArray)
204         @inbounds for item in eachrow(scaffold_feature_sf)
205             scaffold_bitarray[item.Start:item.End,1] = [true for
206             ↵   base in item.Start:item.End]
207             if size(scaffold_bitarray)[2] == 2 #if the bitarray
208             ↵   is stranded
209                 if item.Strand == '+'
210                     scaffold_bitarray[item.Start:item.End,2] =
211                     ↵   [true for base in item.Start:item.End]
212             end
213         end
214     end
215
216     function
217         → subtract_features_from_bitarray!(scaffold_feature_sf :: DataFrame,
218         → scaffold_bitarray :: BitArray)
219         @inbounds for item in eachrow(scaffold_feature_sf)
220             scaffold_bitarray[item.Start:item.End,1] = [false for
221             ↵   base in item.Start:item.End]
222         end
223
224     end
225
226     function get_feature_df_from_bitarray(scaffold_id :: String,
227         → scaffold_bitarray :: BitArray)
228         size(scaffold_bitarray)[2] == 2 ? scaffold_feature_df =
229             ↵   DataFrame(SeqID = String[], Start = Integer[], End =
230             ↵   Integer[], Strand=Char[]) : scaffold_feature_df =
231             ↵   DataFrame(SeqID = String[], Start = Integer[], End =
232             ↵   Integer[])
233
234         new_feature_start = findnext(view(scaffold_bitarray,:,:1),
235             ↵   1)
236         while new_feature_start ≠ nothing # while new features
237             ↵   are still found on the bitarray

```

```

221             if size(scaffold_bitarray)[2] == 2 #if stranded,
222                 ← get strand info
223                 scaffold_bitarray[new_feature_start,2] == 1 ?
224                     ← new_feature_strand = '+' :
225                     ← new_feature_strand = '-'
226             end
227             if
228                 ← findnext(!eval,view(scaffold_bitarray,:,:1),new_feature_start)
229                 ← ≠ nothing
230                 new_feature_end =
231                     ← findnext(!eval,view(scaffold_bitarray,:,:1),new_feature_start)
232                     ← #find next false after feature start and
233                     ← subtract 1 for feature end
234             else
235                 new_feature_end = size(scaffold_bitarray)[1]
236                     ← #if none is found, the end of the feature
237                     ← is the end of hte scaffold
238             end
239             size(scaffold_bitarray)[2] == 2 ?
240                 ← push!(scaffold_feature_df,[scaffold_id,
241                 ← new_feature_start, new_feature_end,
242                 ← new_feature_strand]) :
243                 ← push!(scaffold_feature_df,[scaffold_id,
244                 ← new_feature_start, new_feature_end]) #push
245                 ← stranded feature info as appropriate
246             new_feature_start =
247                 ← findnext(view(scaffold_bitarray,:,:1),
248                 ← new_feature_end+1)
249             end
250         return scaffold_feature_df
251     end

```

15.2.12 /src/genome_sampling/sequence_sampler.jl

```

1 #####SAMPLING FUNCTIONS#####
2 #function for a Distributed worker to produce a set of samples of given
3     ← parameters from genomic sequences
4 function get_sample_set(input_sample_jobs :: RemoteChannel,
5     ← completed_sample_jobs :: RemoteChannel,
6     ← progress_updates :: RemoteChannel)
7     while isready(input_sample_jobs)

```

```

5      genome_path, genome_index_path, partition_df, partitionid,
6      ↵ sample_set_length, sample_window_min, sample_window_max,
7      ↵ deterministic = take!(input_sample_jobs)
8
9
10     stranded::Bool = get_strand_dict()[partitionid]
11    scaffold_sequence_record_dict::Dict{String,LongSequence} =
12      ↵ build_scaffold_seq_dict(genome_path, genome_index_path)
13
14    sample_df =
15      ↵ DataFrame(SampleScaffold=String[], SampleStart=Integer[], SampleEnd=Integer[])
16    metacoordinate_bitarray = trues(partition_df.MetaEnd[end])
17    sample_set_counter = 0
18
19
20    while sample_set_counter < sample_set_length #while we don't yet
21      ↵ have enough sample sequence
22        sample_scaffold::String, sample_Start::Integer,
23          ↵ sample_End::Integer, sample_metaStart::Integer,
24          ↵ sample_metaEnd::Integer, sample_sequence::LongSequence,
25          ↵ strand::Char = get_sample(metacoordinate_bitarray,
26          ↵ sample_window_min, sample_window_max, partition_df,
27          ↵ scaffold_sequence_record_dict; stranded=stranded,
28          ↵ deterministic=deterministic)
29        push!(sample_df,[sample_scaffold, sample_Start, sample_End,
30          ↵ sample_sequence, strand]) #push the sample to the df
31        sample_length = sample_End - sample_Start + 1
32        sample_set_counter += sample_length #increase the counter by
33          ↵ the length of the sampled sequence
34        metacoordinate_bitarray[sample_metaStart:sample_metaEnd] =
35          ↵ [false for base in 1:sample_length] #mark these residues
36          ↵ as sampled
37        put!(progress_updates, (partitionid,
38          ↵ min(sample_set_counter,sample_set_length)))
39
40      end
41      put!(completed_sample_jobs,(partitionid, sample_df))
42    end
43  end
44
45  #get_sample_set() SUBFUNCTIONS
46  #function defining whether partitions respect stranding
47    ↵ upon fetching sequence (ie is the sequence fetched in
48    ↵ the feature strand orientation, or are we agnostic
49    ↵ about the strand we sample?)
50  function get_strand_dict()

```

```

28
29     return
30         ↳ Dict("exon"=>true, "periexonic"=>true, "intergenic"=>false)
31 end
32 #function to obtain a dict of scaffold sequences from a
33     ↳ FASTA reader
34 function build_scaffold_seq_dict(genome_fa, genome_index)
35     genome_reader = open(FASTA.Reader, genome_fa,
36         ↳ index=genome_index)
37     seq_dict::Dict{String, LongSequence} =
38         ↳ Dict{String,FASTA.Record}()
39     @inbounds for record in genome_reader
40         id = FASTA.identifier(record)

41             ↳ seq_dict[rectify_identifier(id)]=FASTA.sequence(record)
42     end
43     close(genome_reader)
44     return seq_dict
45 end

46 # function to convert scaffold ID from that observed by
47     ↳ the masked .fna to the more legible one observed by
48     ↳ the GRCz11 GFF3
49 function rectify_identifier(scaffold_id::String)
50     if length(scaffold_id) ≥ 4 && scaffold_id[1:4] =
51         ↳ "CM00" #marks chromosome scaffold
52         chr_code = scaffold_id[5:10]
53         chr_no = "$(Int((parse(Float64,chr_code)) -
54             ↳ 2884.2))"
55         return chr_no
56     else
57         return scaffold_id
58     end
59 end

60
61 #function to produce a single sample from a
62     ↳ metacoordinate set and the feature df
63 function get_sample(metacoordinate_bitarray::BitArray,
64     ↳ sample_window_min::Integer,
65     ↳ sample_window_max::Integer, partition_df::DataFrame,
66     ↳ scaffold_seq_dict::Dict{String,LongSequence};
67     ↳ stranded::Bool=false, deterministic::Bool=false)
68     proposal_acceptance = false
69     sample_metaStart = 0

```

```

57         sample_metaEnd = 0
58         sample_Start = 0
59         sample_End = 0
60         sample_sequence = LongSequence{DNAAlphabet{2}}("")"
61         sample_scaffold = ""
62         strand = nothing
63
64     while proposal_acceptance == false
65         available_indices =
66             ↳ findall(metacoordinate_bitarray) #find all
67             ↳ unsampled indices
68         window = "FAIL"
69         start_index,feature_metaStart,feature_metaEnd,
70             ↳ feature_length = 0,0,0,0
71         strand = '0'
72         while window == "FAIL"
73             start_index = rand(available_indices)
74                 ↳ #randomly choose from the unsampled
75                 ↳ indices
76             feature_metaStart, feature_metaEnd, strand =
77                 ↳ get_feature_params_from_metacoord(start_index,
78                 ↳ partition_df, stranded)
79             feature_length =
80                 ↳ length(feature_metaStart:feature_metaEnd)
81                 ↳ #find the metaboundaries of the feature
82                 ↳ the index occurs in
83             if feature_length ≥ sample_window_min #don't
84                 ↳ bother finding windows on features
85                 ↳ smaller than min
86             window =
87                 ↳ determine_sample_window(feature_metaStart,
88                 ↳ feature_metaEnd, start_index,
89                 ↳ metacoordinate_bitarray,
90                 ↳ sample_window_min, sample_window_max)
91                 ↳ #get an appropriate sampling window
92                 ↳ around the selected index, given the
93                 ↳ feature boundaries and params
94         end
95     end
96
97     sample_scaffoldid, sample_scaffold_start,
98         ↳ sample_scaffold_end =
99         ↳ meta_to_feature_coord(window[1],window[2],partition_df)

```

```

79
80         proposal_sequence =
81             ↵   fetch_sequence(sample_scaffoldid,
82             ↵   scaffold_seq_dict, sample_scaffold_start,
83             ↵   sample_scaffold_end, strand;
84             ↵   deterministic=deterministic) #get the
85             ↵   sequence associated with the sample window
86
87
88     if mask_check(proposal_sequence)
89
90         proposal_acceptance = true #if the sequence
91             ↵   passes the mask check, accept the
92             ↵   proposed sample
93
94         sample_scaffold = sample_scaffoldid
95         sample_Start = sample_scaffold_start
96         sample_End = sample_scaffold_end
97         sample_metaStart = window[1]
98         sample_metaEnd = window[2]
99         sample_sequence=proposal_sequence
100
101     end
102
103     return sample_scaffold, sample_Start, sample_End,
104         ↵   sample_metaStart, sample_metaEnd,
105         ↵   sample_sequence, strand
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1193
1194
1195
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1205
1206
1207
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1215
1216
1217
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1225
1226
1227
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1235
1236
1237
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1245
1246
1247
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1255
1256
1257
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1265
1266
1267
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1294
1295
1296
1296
1297
1298
1298
1299
1300
1301
1302
1303
1303
1304
1305
1305
1306
1307
1307
1308
1309
1309
1310
1311
1312
1313
1314
1314
1315
1316
1316
1317
1318
1318
1319
1320
1320
1321
1322
1322
1323
1324
1324
1325
1326
1326
1327
1328
1328
1329
1330
1330
1331
1332
1332
1333
1334
1334
1335
1336
1336
1337
1338
1338
1339
1340
1340
1341
1342
1342
1343
1344
1344
1345
1346
1346
1347
1348
1348
1349
1350
1350
1351
1352
1352
1353
1354
1354
1355
1356
1356
1357
1358
1358
1359
1360
1360
1361
1362
1362
1363
1364
1364
1365
1366
1366
1367
1368
1368
1369
1370
1370
1371
1372
1372
1373
1374
1374
1375
1376
1376
1377
1378
1378
1379
1380
1380
1381
1382
1382
1383
1384
1384
1385
1386
1386
1387
1388
1388
1389
1390
1390
1391
1392
1392
1393
1394
1394
1395
1396
1396
1397
1398
1398
1399
1400
1400
1401
1402
1402
1403
1404
1404
1405
1406
1406
1407
1408
1408
1409
1410
1410
1411
1412
1412
1413
1414
1414
1415
1416
1416
1417
1418
1418
1419
1420
1420
1421
1422
1422
1423
1424
1424
1425
1426
1426
1427
1428
1428
1429
1430
1430
1431
1432
1432
1433
1434
1434
1435
1436
1436
1437
1438
1438
1439
1440
1440
1441
1442
1442
1443
1444
1444
1445
1446
1446
1447
1448
1448
1449
1450
1450
1451
1452
1452
1453
1454
1454
1455
1456
1456
1457
1458
1458
1459
1460
1460
1461
1462
1462
1463
1464
1464
1465
1466
1466
1467
1468
1468
1469
1470
1470
1471
1472
1472
1473
1474
1474
1475
1476
1476
1477
1478
1478
1479
1480
1480
1481
1482
1482
1483
1484
1484
1485
1486
1486
1487
1488
1488
1489
1490
1490
1491
1492
1492
1493
1494
1494
1495
1496
1496
1497
1498
1498
1499
1500
1500
1501
1502
1502
1503
1504
1504
1505
1506
1506
1507
1508
1508
1509
1510
1510
1511
1512
1512
1513
1514
1514
1515
1516
1516
1517
1518
1518
1519
1520
1520
1521
1522
1522
1523
1524
1524
1525
1526
1526
1527
1528
1528
1529
1530
1530
1531
1532
1532
1533
1534
1534
1535
1536
1536
1537
1538
1538
1539
1540
1540
1541
1542
1542
1543
1544
1544
1545
1546
1546
1547
1548
1548
1549
1550
1550
1551
1552
1552
1553
1554
1554
1555
1556
1556
1557
1558
1558
1559
1560
1560
1561
1562
1562
1563
1564
1564
1565
1566
1566
1567
1568
1568
1569
1570
1570
1571
1572
1572
1573
1574
1574
1575
1576
1576
1577
1578
1578
1579
1580
1580
1581
1582
1582
1583
1584
1584
1585
1586
1586
1587
1588
1588
1589
1590
1590
1591
1592
1592
1593
1594
1594
1595
1596
1596
1597
1598
1598
1599
1600
1600
1601
1602
1602
1603
1604
1604
1605
1606
1606
1607
1608
1608
1609
1610
1610
1611
1612
1612
1613
1614
1614
1615
1616
1616
1617
1618
1618
1619
1620
1620
1621
1622
1622
1623
1624
1624
1625
1626
1626
1627
1628
1628
1629
1630
1630
1631
1632
1632
1633
1634
1634
1635
1636
1636
1637
1638
1638
1639
1640
1640
1641
1642
1642
1643
1644
1644
1645
1646
1646
1647
1648
1648
1649
1650
1650
1651
1652
1652
1653
1654
1654
1655
1656
1656
1657
1658
1658
1659
1660
1660
1661
1662
1662
1663
1664
1664
1665
1666
1666
1667
1668
1668
1669
1670
1670
1671
1672
1672
1673
1674
1674
1675
1676
1676
1677
1678
1678
1679
1680
1680
1681
1682
1682
1683
1684
1684
1685
1686
1686
1687
1688
1688
1689
1690
1690
1691
1692
1692
1693
1694
1694
1695
1696
1696
1697
1698
1698
1699
1700
1700
1701
1702
1702
1703
1704
1704
1705
1706
1706
1707
1708
1708
1709
1710
1710
1711
1712
1712
1713
1714
1714
1715
1716
1716
1717
1718
1718
1719
1720
1720
1721
1722
1722
1723
1724
1724
1725
1726
1726
1727
1728
1728
1729
1730
1730
1731
1732
1732
1733
1734
1734
1735
1736
1736
1737
1738
1738
1739
1740
1740
1741
1742
1742
1743
1744
1744
1745
1746
1746
1747
1748
1748
1749
1750
1750
1751
1752
1752
1753
1754
1754
1755
1756
1756
1757
1758
1758
1759
1760
1760
1761
1762
1762
1763
1764
1764
1765
1766
1766
1767
1768
1768
1769
1770
1770
1771
1772
1772
1773
1774
1774
1775
1776
1776
1777
1778
1778
1779
1780
1780
1781
1782
1782
1783
1784
1784
1785
1786
1786
1787
1788
1788
1789
1790
1790
1791
1792
1792
1793
1794
1794
1795
1796
1796
1797
1798
1798
1799
1800
1800
1801
1802
1802
1803
1804
1804
1805
1806
1806
1807
1808
1808
1809
1810
1810
1811
1812
1812
1813
1814
1814
1815
1816
1816
1817
1818
1818
1819
1820
1820
1821
1822
1822
1823
1824
1824
1825
1826
1826
1827
1828
1828
1829
1830
1830
1831
1832
1832
1833
1834
1834
1835
1836
1836
1837
1838
1838
1839
1840
1840
1841
1842
1842
1843
1844
1844
1845
1846
1846
1847
1848
1848
1849
1850
1850
1851
1852
1852
1853
1854
1854
1855
1856
1856
1857
1858
1858
1859
1860
1860
1861
1862
1862
1863
1864
1864
1865
1866
1866
1867
1868
1868
1869
1870
1870
1871
1872
1872
1873
1874
1874
1875
1876
1876
1877
1878
1878
1879
1880
1880
1881
1882
1882
1883
1884
1884
1885
1886
1886
1887
1888
1888
1889
1890
1890
1891
1892
1892
1893
1894
1894
1895
1896
1896
1897
1898
1898
1899
1900
1900
1901
1902
1902
1903
1904
1904
1905
1906
1906
1907
1908
1908
1909
1910
1910
1911
1912
1912
1913
1914
1914
1915
1916
1916
1917
1918
1918
1919
1920
1920
1921
1922
1922
1923
1924
1924
1925
1926
1926
1927
1928
1928
1929
1930
1930
1931
1932
1932
1933
1934
1934
1935
1936
1936
1937
1938
1938
1939
1940
1940
1941
1942
1942
1943
1944
1944
1945
1946
1946
1947
1948
1948
1949
1950
1950
1951
1952
1952
1953
1954
1954
1955
1956
1956
1957
1958
1958
1959
1960
1960
1961
1962
1962
1963
1964
1964
1965
1966
1966
1967
1968
1968
1969
1970
1970
1971
1972
1972
1973
1974
1974
1975
1976
1976
1977
1978
1978
1979
1980
1980
1981
1982
1982
1983
1984
1984
1985
1986
1986
1987
1988
1988
1989
1990
1990
1991
1992
1992
1993
1994
1994
1995
1996
1996
1997
1998
1998
1999
2000
2000
2001
2002
2002
2003
2004
2004
2005
2006
2006
2007
2008
2008
2009
2010
2010
2011
2012
2012
2013
2014
2014
2015
2016
2016
2017
2018
2018
2019
2020
2020
2021
2022
2022
2023
2024
2024
2025
2026
2026
2027
2028
2028
2029
2030
2030
2031
2032
2032
2033
2034
2034
2035
2036
2036
2037
2038
2038
2039
2040
2040
2041
2042
2042
2043
2044
2044
2045
2046
2046
2047
2048
2048
2049
2050
2050
2051
2052
2052
2053
2054
2054
2055
2056
2056
2057
2058
2058
2059
2060
2060
2061
2062
2062
2063
2064
2064
2065
2066
2066
2067
2068
2068
2069
2070
2070
2071
2072
2072
2073
2074
2074
2075
2076
2076
2077
2078
2078
2079
2080
2080
2081
2082
2082
2083
2084
2084
2085
2086
2086
2087
2088
2088
2089
2090
2090
2091
2092
2092
2093
2094
2094
2095
2096
2096
2097
2098
2098
2099
2100
2100
2101
2102
2102
2103
2104
2104
2105
2106
2106
2107
2108
2108
2109
2110
2110
2111
2112
2112
2113
2114
2114
2115
2116
2116
2117
2118
2118
2119
2120
2120
2121
2122
2122
2123
2124
2124
2125
2126
2126
2127
2128
2128
2129
2130
2130
2131
2132
2132
2133
2134
2134
2135
2136
2136
2137
2138
2138
2139
2140
2140
2141
2142
2142
2143
2144
2144
2145
2146
2146
2147
2148
2148
2149
2150
2150
2151
2152
2152
2153
2154
2154
2155
2156
2156
2157
2158
2158
2159
2160
2160
2161
2162
2162
2163
2164
2164
2165
2166
2166
2
```

```

103         window_start = feature_metaStart
104         window_end = feature_metaEnd
105         return window_start, window_end
106     else #if this fails, build the biggest window we can
107         ↳ from the sampling point, within the feature
108         featurepos = metacoord - feature_metaStart + 1
109         next_sampled_index = findnext(!eval,
110             ↳ metacoord_bitarray[feature_metaStart:feature_metaEnd],
111             ↳ featurepos)
112         prev_sampled_index = findprev(!eval,
113             ↳ metacoord_bitarray[feature_metaStart:feature_metaEnd],
114             ↳ featurepos)
115         next_sampled_index == nothing ? (window_end =
116             ↳ feature_metaEnd) : (window_end =
117             ↳ next_sampled_index + feature_metaStart - 1)
118         prev_sampled_index == nothing ? (window_start =
119             ↳ feature_metaStart) : (window_start =
120             ↳ prev_sampled_index + feature_metaStart - 1)
121         windowsize = window_end - window_start + 1
122         #check to see if this window is bigger than the
123         ↳ min, if not, return a failure code
124         windowsize < sample_window_min && return "FAIL"
125         #check to see if this window is bigger than the
126         ↳ max, if so, trim it before returning,
127         ↳ removing as evenly as possible around the
128         ↳ metacoordinate
129         if windowsize > sample_window_max
130             bases_to_trim = windowsize -
131                 ↳ sample_window_max
132             clearance_5P = metacoord - window_start + 1
133             clearance_3P = window_end - metacoord + 1
134             trimmed=0
135             while trimmed < bases_to_trim
136                 if clearance_5P ≥ clearance_3P &&
137                     ↳ clearance_5P > 0
138                     clearance_5P -= 1
139                 elseif clearance_3P > clearance_5P &&
140                     ↳ clearance_3P > 0
141                     clearance_3P -= 1
142             end
143             trimmed +=1
144         end
145         window_start = metacoord - clearance_5P + 1

```

```

130                     window_end = metacoord + clearance_3P - 1
131                 end
132             return window_start, window_end
133         end
134     end
135
136
137     # function to check for repetitive stretches or
138     # degenerate bases in proposal sequence
139     function mask_check(proposal_sequence::LongSequence)
140         proposal_acceptance = true
141         if hasambiguity(proposal_sequence) ||
142             isrepetitive(proposal_sequence,
143             (length(proposal_sequence) ÷ 10))
144             proposal_acceptance = false
145         end
146         return proposal_acceptance
147     end
148
149
150 ######SHARED SEQUENCE FETCHER#####
151 # function to get proposal sequence from dict of scaffold sequences,
152 # given coords and scaffold id
153 function fetch_sequence(scaffold_id::String,
154     → scaffold_seq_dict::Dict{String, LongSequence},
155     → proposal_start::Integer, proposal_end::Integer, strand::Char;
156     → deterministic=false)
157     if strand == '0' #unstranded samples may be returned with no
158         → preference in either orientation
159         deterministic ? strand = '+' : (rand(1)[1] ≤ .5 ? strand = '+' :
160             → strand = '-')
161     end
162     if strand == '+'
163         proposal_sequence =
164             → scaffold_seq_dict[scaffold_id][proposal_start:proposal_end]
165     elseif strand == '-'
166         proposal_sequence =
167             → reverse_complement(scaffold_seq_dict[scaffold_id][proposal_start:proposal_end])
168     else
169         throw(ArgumentError("Invalid sample code! Must be '+', '-' , or
170             → '0' (random strand)"))
171     end
172
173     return proposal_sequence

```

```

161 end

162

163 #####SHARED BASIC COORDINATE SUBFUNCTIONS#####
164 # function to push scaffold ID, start, and end points of given
165 #       ↳ featuretype to supplied dataframe
166 function build_feature_df!(GFF3_path::String, feature_type::String,
167   ↳ scaffold_exclusion::String, feature_df::DataFrame)
168   reader = open(GFF3.Reader, GFF3_path) # access the GFF3
169   @inbounds for record in reader # iterate over Gff3 records
170     if GFF3.isfeature(record) # if the record is a feature
171       if GFF3.featuretype(record) == feature_type #if the features
172         ↳ is of the requested type, get the following info
173         seqID = GFF3.seqid(record)
174         if seqID != scaffold_exclusion
175           seq_start = GFF3.seqstart(record)
176           seq_end = GFF3.seqend(record)
177           if feature_type == "CDS" || feature_type == "gene"
178             seq_strand = convert(Char, GFF3.strand(record))
179             push!(feature_df, [seqID, seq_start, seq_end,
180             ↳ seq_strand])
181           else
182             push!(feature_df, [seqID, seq_start, seq_end]) #
183             ↳ push relevant info to the df
184           end
185         end
186       end
187     end
188   close(reader)
189 end

190

191 #function to assemble dataframe of scaffold coords + metacoords given
192 #       ↳ gff3
193 function build_scaffold_df(gff3_path)
194   scaffold_df = DataFrame(SeqID = String[], Start = Integer[], End =
195   ↳ Integer[])
196   build_feature_df!(gff3_path, "supercontig", "MT", scaffold_df)
197   build_feature_df!(gff3_path, "chromosome", "MT", scaffold_df)
198   add_metacoordinates!(scaffold_df)
199   return scaffold_df
200 end

201

202 #function to add pad to either side of some featureset

```

```

197 function add_pad_to_coordinates!(feature_df::DataFrame,
198   → scaffold_df::DataFrame, pad_size::Integer;
199   → col_symbols::Array{Symbol}=[:Start, :End])
200   pad_start_array = zeros(Integer,size(feature_df,1))
201   pad_end_array = zeros(Integer,size(feature_df,1))
202   feature_df[!, col_symbols[1]] = [max(feature_df.Start[i]-pad_size,1)
203     → for i in 1:size(feature_df,1)] #truncate pads at beginning and
204     → end of scaffolds
205   feature_df[!, col_symbols[2]] =
206     → [min(feature_df.End[i]+pad_size,scaffold_df.End[findfirst(isequal(feature_df.
207       → .Start, scaffold_df.Start))],1)
208     → for i in 1:size(feature_df,1)]
209 end
210
211 ####SHARED METACOORDINATE FUNCTIONS####
212 # function to add a metacoordinate column to a dataframe of scaffold
213   → positions, allowing sampling across all scaffolds
214 function add_metacoordinates!(feature_df::DataFrame)
215   meta_start_array = Integer[]
216   meta_end_array = Integer[]
217   metaposition = 1
218   @inbounds for feature in eachrow(feature_df)
219     push!(meta_start_array, (metaposition))
220     metaposition += feature.End - feature.Start
221     push!(meta_end_array, (metaposition))
222     metaposition += 1
223   end
224   feature_df.MetaStart = meta_start_array # metacoordinate contains
225     → 'start' metaposition across all genomic material
226   feature_df.MetaEnd = meta_end_array # metacoordinate contains 'end'
227     → metaposition across all genomic material
228 end
229
230 # function to convert metacoordinate set to scaffold-relative coordinates
231 function meta_to_feature_coord(meta_start::Integer, meta_end::Integer,
232   → feature_df::DataFrame)
233   feature_row = get_feature_row_index(feature_df, meta_start)
234   seqid = feature_df.SeqID[feature_row]
235   scaffold_start = feature_df.Start[feature_row] + (meta_start -
236     → feature_df.MetaStart[feature_row])
237   scaffold_end = feature_df.End[feature_row] -
238     → (feature_df.MetaEnd[feature_row] - meta_end)
239   return seqid, scaffold_start, scaffold_end
240 end

```

```

228
229 #function to obtain the feature boundaries and strand of the feature that
  ↳ a metacoordinate falls within
230 function get_feature_params_from_metacoord(metacoordinate::Integer,
  ↳ feature_df::DataFrame, stranded::Bool)
231     feature_row = get_feature_row_index(feature_df, metacoordinate)
232     feature_metaStart = feature_df.MetaStart[feature_row]
233     feature_metaEnd = feature_df.MetaEnd[feature_row]
234     stranded ? feature_strand = feature_df.Strand[feature_row] :
  ↳ feature_strand = '0'
235     return feature_metaStart, feature_metaEnd, feature_strand
236 end
237
238 #function obtain the index of a feature given its metacoordinate
239 function get_feature_row_index(feature_df::DataFrame,
  ↳ metacoordinate::Integer)
240     total_feature_bases = feature_df.MetaEnd[end]
241     if metacoordinate == total_feature_bases #edge case of metacoord at
  ↳ end of range
        return size(feature_df,1) #last index in df
242     else
243         index =
244             findfirst(end_coord→end_coord>metacoordinate, feature_df.MetaEnd)
  ↳ #find the index of the feature whose end metacoord is > the
  ↳ query metacoord
245         if index > 1 && metacoordinate == feature_df.MetaEnd[index-1] #if
  ↳ the metacoordinate is the last base of the previous feature
246             if metacoordinate ≥ feature_df.MetaStart[index-1] &&
  ↳ metacoordinate ≤ feature_df.MetaEnd[index-1] #confirm
  ↳ that the metacoord is in the previous feature
                  return index-1 #return the previous feature index
247             end
248         elseif metacoordinate ≥ feature_df.MetaStart[index] &&
  ↳ metacoordinate ≤ feature_df.MetaEnd[index] #else confirm
  ↳ that the metacoordinate is in the found feature
                  return index #return the found feature index
249     else
250         throw(DomainError("Unexpected metacoordinate $metacoordinate
  ↳ in partition of $total_feature_bases bases, with feature
  ↳ start $(feature_df.MetaStart[index]), end
  ↳ $(feature_df.MetaEnd[index]))")
251     end
252 end
253
254 end

```

 255 end

15.2.13 /src/likelihood_funcs/bg_lh_matrix.jl

```

1 #function to obtain positional likelihoods for a sequence under a given
2   ↳ BHMM.
3 function get_BGHMM_symbol_lh(seq::AbstractMatrix, hmm::BHMM)
4   @assert size(seq)[1] == 1
5   (seq=Array(transpose(seq))) # one sequence at a time only
6   symbol_lhs = zeros(length(seq))
7   length_mask = [length(seq)-1]
8
9   lls = bw_llhs(hmm, seq) #obtain log likelihoods for sequences and
10    ↳ states
11  log_α = messages_forwards_log(hmm.a, hmm.A, lls, length_mask) #get
12    ↳ forward messages
13  log_β = messages_backwards_log(hmm.A, lls, length_mask) # get
14    ↳ backwards messages
15
16  #calculate observation probability and γ weights
17  K,Tmaxplus1,Strand = size(lls) #the last T value is the 0 end marker
18    ↳ of the longest T
19
20  #transforms to cut down log_ξ, log_γ assignment times
21  lls = permutedims(lls, [2,1,3]) # from (K,T) to (T,K)
22  log_α = permutedims(log_α, [2,1,3])
23  log_β = permutedims(log_β, [2,1,3])
24
25  log_γ = fill(-Inf, Tmaxplus1,K)
26  log_pobs = logsumexp(lps.(log_α[1,:], log_β[1,:]))
27
28  @inbounds for i = 1:K, t = 1:length_mask[1]
29      log_γ[t,i] = lps(log_α[t,i],log_β[t,i],-log_pobs)
30  end
31
32  for t in 1:length_mask[1]
33      symbol_lh::AbstractFloat = -Inf #ie log(p=0)
34      for k = 1:K #iterate over states
35          state_symbol_lh::AbstractFloat = lps(log_γ[t,k],
36            ↳ log(hmm.B[k].p[seq[t]])) #state symbol likelihood is
37            ↳ the γ weight * the state symbol probability (log
38            ↳ implementation)

```

```

31         symbol_lh = logaddexp(symbol_lh, state_symbol_lh) #sum
32             ↵ the probabilities over states
33     end
34     symbol_lhs[t] = symbol_lh
35   end
36
37   return symbol_lhs[1:end-1] #remove trailing index position
38 end
39
40 #function to calculate BGHMM from an observation set and a dict of BGHMMs
41 function BGHMM_likelihood_calc(observations::DataFrame, BGHMM_dict::Dict,
42   ↵ code_partition_dict = get_partition_code_dict(false); symbol=:PadSeq)
43   lh_matrix_size = ((findmax(length.(collect(values(observations[!,,
44   ↵ symbol]))))[1]), length(observations[!, symbol]))
45   BGHMM_lh_matrix = zeros(lh_matrix_size) #T, Strand, 0
46
47   #@showprogress 1 "Writing frags to matrix.."
48   Threads.@threads for (jobid, frag) in BGHMM_fragments
49     (frag_start, o, partition, strand) = jobid
50
51     partition_BGHMM::BHMM =
52       ↵ BGHMM_dict[code_partition_dict[partition]]
53     no_symbols = length(partition_BGHMM.B[1].p)
54     order = Int(log(4,no_symbols) - 1)
55
56     order_seq = get_order_n_seqs([frag], order)
57     coded_seq = code_seqs(order_seq)
58
59     subseq_symbol_lh = get_BGHMM_symbol_lh(coded_seq,
60       ↵ partition_BGHMM)
61
62     if strand == -1
63       subseq_symbol_lh = reverse(subseq_symbol_lh)
64     end #positive, unstranded frags are inserted as-is
65     BGHMM_lh_matrix[frag_start:frag_start+length(frag)-1,o] =
66       ↵ subseq_symbol_lh
67   end
68
69   return BGHMM_lh_matrix

```

```

67 end

68

69 function fragment_observations_by_BGHMM(seqs :: AbstractVector,
70   ↳ masks :: AbstractVector)
71   likelihood_jobs = Vector{Tuple{Tuple, LongSequence{DNAAlphabet{2}}}}()
72   @showprogress 1 "Fragmenting observations by partition ... " for (o,
73     ↳ obs_seq) in enumerate(seqs)
74     mask = masks[o]
75     frags = Vector{LongSequence{DNAAlphabet{2}}}()

76     frag = LongSequence{DNAAlphabet{2}}()

77     frag_end=0
78     frag_start = 1

79     while frag_start < length(obs_seq) # while we're not at the
80       ↳ sequence end
81       curr_partition = mask[frag_start,1] #get the partition code
82       ↳ of the frag start
83       curr_strand = mask[frag_start,2] #get the strand of the frag
84       ↳ start

85       #JOBID COMPOSED HERE
86       jobid = (frag_start, o, curr_partition, curr_strand) #compose
87       ↳ an identifying index for this frag

88       findnext(!isequal(curr_partition),mask[:,1],frag_start) ≠
89         ↳ nothing ? frag_end =
90         ↳ findnext(!isequal(curr_partition),mask[:,1],frag_start)
91         ↳ -1 : frag_end = length(obs_seq) #find the next position
92         ↳ in the frag that has a different partition mask value
93         ↳ from hte current one and set that position-1 to frag end,
94         ↳ alternately frag end is end of the overall sequence
95       frag = obs_seq[frag_start:frag_end] #get the frag bases
96       if curr_strand == -1 #if the fragment is reverse stranded
97         ↳ reverse_complement!(frag) #use the reverse complement
98         ↳ sequence
99       end

100      push!(likelihood_jobs,(jobid, frag)) #put the frag in the
101      ↳ jobs vec
102      frag_start = frag_end + 1 #move on
103    end

```

```

95     end
96     return likelihood_jobs
97 end

```

15.2.14 /src/likelihood_funcs/hmm.jl

```

1 function obs_lh_given_hmm(observations, hmm; linear=true)
2     linear ? (return linear_likelihood(observations, hmm)) : (return
3         ↳ bw_likelihood(Matrix(transpose(observations)), hmm))
4 end
5
6 function bw_likelihood(observations, hmm)
7     obs_lengths = [findfirst(iszero, observations[:,o])-1 for o in
8         ↳ 1:size(observations,2)]
9
10    lls = bw_llhs(hmm, observations)
11    log_α = messages_forwards_log(hmm.a, hmm.A, lls, obs_lengths)
12    log_β = messages_backwards_log(hmm.A, lls, obs_lengths)
13
14    O = size(lls)[3] #the last T value is the 0 end marker of the longest
15    ↳ T
16    log_pobs = zeros(O)
17    Threads.@threads for o in 1:O
18        log_pobs[o] = logsumexp(lps.(log_α[:,1,o], log_β[:,1,o]))
19    end
20
21    return sum(log_pobs)
22 end
23
24 function linear_likelihood(observations,hmm)
25     O= size(observations,1);
26     obs_lengths = [findfirst(iszero,observations[o,:])-1 for o in
27         ↳ 1:size(observations,1)] #mask calculations here rather than
28         ↳ mle_step to prevent recalculation every iterate
29
30     a = transpose(log.(hmm.a)); A = log.(hmm.A)
31     N = length(hmm.B)
32     mask=observations.≠0
33     #INITIALIZATION
34     βoi_T = zeros(0,N); βoi_t = zeros(0,N) #log betas at T initialised as
35         ↳ zeros

```

```

31     #RECURRENCE
32     βoi_T = backwards_lh_sweep!(hmm, A, N, βoi_T, βoi_t, observations,
33                               → mask, obs_lengths)
34
35     #TERMINATION
36     lls = c_llhs(hmm,observations[:,1])
37     α1om = lls .+ a #first position forward msgs
38
39     return lps([logsumexp(lps.(α1om[o,:], βoi_T[o,:])) for o in 1:0])
40 end
41
42     #LINEAR_STEP SUBFUNCS
43     function backwards_lh_sweep!(hmm, A, N, βoi_T, βoi_t,
44                               → observations, mask, obs_lengths)
45         for t in maximum(obs_lengths)-1:-1:1
46             last_β=copy(βoi_T)
47             lls = c_llhs(hmm,observations[:,t+1])
48             omask = findall(mask[:,t+1])
49             βoi_T[omask,:] .+= view(lls,omask,:)
50             Threads.@threads for m in 1:N
51                 βoi_t[omask,m] =
52                     → logsumexp.(eachrow(view(βoi_T,omask,:).+transpose(view(
53                         end
54                         βoi_T=copy(βoi_t)
55
56                         end
57                         return βoi_T
58
59                         end

```

15.2.15 /src/reports/chain_report.jl

```

1 struct Chain_Report
2     id::Chain_ID
3     final_hmm::BHMM
4     test_lh::AbstractFloat
5     naive_lh::AbstractFloat
6     final_delta::AbstractFloat
7     state_run_lengths::AbstractVector{AbstractFloat}
8     convergence_values::Chains
9     convergence_diagnostic::ChainDataFrame
10    converged::Bool
11 end
12
13 function Base.show(io::IO, report::Chain_Report)

```

```

14 nominal_dict=Dict(0=>"th",1=>"st",2=>"nd",3=>"rd",4=>"th",5=>"th")
15 haskey(nominal_dict,report.id.order) ?
16   → (nom_str=nominal_dict[report.id.order]) : (nom_str="th")
17 println(io, "BHMM EM Chain Results\n"; bold=true)
18 println(io, "$(report.id.K)-state, $(report.id.order)$nom_str order
19   → BHMM")
20 println(io, "Trained on observation set \$(report.id.obs_id)\n")
21 report.test_lh > report.naive_lh ? (lh_str="greater";
22   → lh_color=:green) : (lh_str="less"; lh_color=:red)
23 println(io, "Test logP(0|θ): $(report.test_lh), $lh_str than the
24   → naive model's $(report.naive_lh)\n"; color=lh_color)
25 println(io, "Replicate $(report.id.replicate)")
26 report.converged ? println(io, "Converged with final step δ
27   → $(report.final_delta)") : println(io, "Failed to converge!")
28 println(io, "-----")
29   → -----
30   → -----
31   → -----
32   → -----
33   → -----
34   → zidx=findfirst(!iszero,report.convergence_values["logP(0|θ)"].data)[1]
35   → lh_vec=Vector([report.convergence_values["logP(0|θ)"].data ... ][zidx:end])
36   → lh_plot=lineplot([zidx:length(lh_vec)+zidx-1 ... ],lh_vec;title="Chain
37   → likelihood evolution", xlabel="Training iterate",
38   → xlim=(0,length(lh_vec)+zidx+1), ylim=
39   → (floor(minimum(lh_vec),sigdigits=2),ceil(maximum(lh_vec),sigdigits=2)),
40   → name="logP(0|θ)")
41 lineplot!(lh_plot,[report.naive_lh for i in 1:length(lh_vec)],
42   → color=:magenta,name="naive")
43 show(io, lh_plot)
44 println(io, "\n")
45 k1vec=Vector([report.convergence_values["K1"].data ... ])

```

```

40     k_plot=lineplot(k1vec, title="State p(Auto) evolution",
41         ← xlabel="Training iterate", ylabel="prob",
42         ← name="K1",ylim=(0,1),xlim=(0,length(k1vec)))
43     for k in 2:report.id.K
44         kvec=Vector([report.convergence_values["K$k"].data ... ])
45         lineplot!(k_plot,kvec,name="K$k")
46     end
47     show(io,k_plot)
48     println(io)
49
50     printstyled(io, "\nConvergence Diagnostics\n",bold=true)
51     if report.convergence_diagnostic.name == "short"
52         printstyled(io, "Convergence diagnostics unavailable for chains
53             ← <10 steps!\n", color=:yellow)
54     elseif report.convergence_diagnostic.name == "error"
55         printstyled(io, "Convergence diagnostics errored! Zeros in
56             ← autotransition matrix?\n", color=:red)
57     else
58         all(Bool.(report.convergence_diagnostic.nt.stationarity)) &&
59             all(Bool.(report.convergence_diagnostic.nt.test)) ?
60                 printstyled(io, "Likelihood and autotransition probabilites
61                     ← converged and passing tests.\n", color=:green) :
62                     printstyled(io, "Not all parameters converged or passing
63                         ← tests!\n",color=:red)
64         display(report.convergence_diagnostic)
65     end
66   end
67 end
68
69 function latex_report(report::Chain_Report)
70
71 end
72
73 function report_chains(chains::Dict{Chain_ID,Vector{EM_step}},
74     ← test_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}});
75     ← naive_hmm=BHMM(ones(1,1),[Categorical(4)])
76     job_ids=[id for (id,chain) in chains] # make list of chain_ids
77     partitions=unique([id.obs_id for id in job_ids]) #get list of unique
78         ← partitions represented among chain_ids
79     naive_order = Integer(log(4,length(naive_hmm.B[1].support))-1)
80     for partition in partitions #check that all partitions are available
81         ← for testing and make sure codes for naive hmm are generated by
82         ← adding job_ids

```

```

68     !(partition in keys(test_sets)) &&
69         → throw(ArgumentError("Observation set $partition required by
70             → chains for testing not present in test sets!"))
71
72     push!(job_ids,
73         → Chain_ID(partition,length(naive_hmm.a),naive_order, 1))
74 end
75
76 code_dict=code_job_obs(job_ids, test_sets) #code all the obs sets
77     → that are necessary ot perform our tests
78 naive_lhs=Dict{String,Float64}() #construct dict of naive likelihoods
79     → for all partitions to be tested prior to individual tests
80 @showprogress 1 "Testing naive hmm..." for
81     → ((partition,order),obs_set) in code_dict
82     order=naive_order &&
83         → (naive_lhs[partition]=obs_lh_given_hmm(code_dict[(partition,naive_order)])
84 end
85
86 reports=Dict{Chain_ID,Chain_Report}()
87 @showprogress 1 "Testing chains..." for (id, chain) in chains
88     chains_array=zeros(length(chain),1+id.K)
89     for (n,step) in enumerate(chain)
90         chains_array[n,1]=step.log_p
91         chains_array[n,2:end]=get_diagonal_array(step.hmm)
92     end
93     convergence_values=Chains(chains_array,[ "logP(0|θ)",
94         → ["K"*string(i) for i in 1:id.K] ... ])
95     if length(chain) ≥ 10
96         try
97             convergence_diagnostic=heideldiag(convergence_values)[1]
98         catch
99             → convergence_diagnostic=ChainDataFrame("error",(:;a⇒[1.]))
100         end
101     else
102         convergence_diagnostic=ChainDataFrame("short",(:;a⇒[1.]))
103     end
104
105     id.K > 1 ?
106         → (mean_run_lengths=sim_run_lengths(get_diagonal_array(chain[end].hmm),1000
107         → : (mean_run_lengths=[Inf])
108         test_lh=obs_lh_given_hmm(code_dict[(id.obs_id,id.order)],
109         → chain[end].hmm)

```

```

98     reports[id]=Chain_Report(id, chain[end].hmm, test_lh,
99         ← naive_lhs[id.obs_id], chain[end].delta, mean_run_lengths,
100        ← convergence_values, convergence_diagnostic,
101        ← chain[end].converged)
102    end
103
104   return reports
105 end
106
107
108 function get_diagonal_array(hmm::BHMM)
109     k = length(hmm.a)
110     diagonal = zeros(k)
111     @inbounds for i in 1:k
112         diagonal[i] = hmm.A[i,i]
113     end
114     return diagonal
115 end
116
117 #function to simulate run lengths for vector of diagonal values
118 function sim_run_lengths(diagonal_value::AbstractArray, samples::Integer)
119     mean_run_lengths = zeros(length(diagonal_value))
120     for (i, value) in enumerate(diagonal_value)
121         if value == 0.
122             mean_run_lengths[i]=0.
123         elseif value ==1.
124             mean_run_lengths[i]=Inf
125         else
126             runlengths = zeros(Integer, samples)
127             for s in 1:samples
128                 run = true
129                 runlength = 0
130                 while run
131                     runlength += 1
132                     if rand(1)[1] > value
133                         run = false
134                     end
135                 end
136                 runlengths[s] = runlength
137             end
138             mean_run_lengths[i] = mean(runlengths)
139         end
140     end
141     return mean_run_lengths

```

138 end

15.2.16 /src/reports/partition_report.jl

```

1 struct Order_Report
2     converged_K::Vector{Float64}
3     converged_lh::Vector{Float64}
4     failed_K::Vector{Float64}
5     failed_lh::Vector{Float64}
6 end
7
8 struct Partition_Report
9     partition_id::String
10    naive_lh::Float64
11    orddict::Dict{Integer,Order_Report}
12    best_model::Tuple{Chain_ID,BHMM}
13    best_repset::Vector{Chain_ID}
14 end
15
16 function Base.show(io::IO, report::Partition_Report)
17     printstyled(io, "BACKGROUND HMM TRAINING REPORT\n", bold=true)
18     printstyled(io, "Genome partition id: $(report.partition_id)\n",
19                 → bold=true, color=:magenta)
20
21     for (order, ordreport) in report.orddict
22         if length(ordreport.failed_lh)>0
23             ordplot=scatterplot(ordreport.converged_K,
24                                 → ordreport.converged_lh; name="Converged", title="Order
25                                 → $order HMMs", xlabel="K# States", ylabel="P(O|θ)",
26                                 → color=:green, xlim=(1,maximum(ordreport.converged_K)),
27                                 → ylim=(floor(min(minimum(ordreport.converged_lh),minimum(ordreport.fai
28             length(ordreport.failed_K) ≥ 1 && scatterplot!(ordplot,
29                     → ordreport.failed_K, ordreport.failed_lh;
30                     → name="Unconverged",color=:red)
31             lineplot!(ordplot, [report.naive_lh for i in
32                             → 1:maximum(ordreport.converged_K)],
33                             → name="Naive",color=:magenta)
34             show(ordplot)
35             println()
36         else
37

```

```

28         ordplot=scatterplot(ordreport.converged_K,
29             ←  ordreport.converged_lh; name="Converged", title="Order
30             $order HMMs", xlabel="K# States", ylabel="P(O|θ)",
31             ←  color=:green, xlim=(1,maximum(ordreport.converged_K)),
32             ←  ylim=(floor(min(minimum(ordreport.converged_lh),report.naive_lh),sigd
33             lineplot!(ordplot, [report.naive_lh for i in
34                 ←  1:maximum(ordreport.converged_K)],
35                 ←  name="Naive",color=:magenta)
36             show(ordplot)
37             println()
38         end
39     end
40 end
41
42 function report_partitions(chain_reports :: Dict{Chain_ID,Chain_Report})
43     partitions=Vector{String}()
44     orders=Vector{Integer}()
45     Ks=Vector{Integer}()
46     reps=Vector{Integer}()
47     for id in keys(chain_reports)
48         !in(id.obs_id, partitions) && push!(partitions, id.obs_id)
49         !in(id.order, orders) && push!(orders, id.order)
50         !in(id.K, Ks) && push!(Ks, id.K)
51         !in(id.replicate, reps) && push!(reps, id.replicate)
52     end
53
54     reports=Vector{Partition_Report}()
55     for partition in partitions
56         best_lh,best_rep=-Inf,Chain_ID("empty",1,0,1)
57         naive_lh=1.
58         orddict=Dict{Integer,Order_Report}()
59         for order in orders
60             conv_statevec=Vector{Float64}()
61             fail_statevec=Vector{Float64}()
62             conv_lh_vec=Vector{Float64}()
63             fail_lh_vec=Vector{Float64}()
64             for (id,report) in chain_reports
65                 if id.obs_id==partition && id.order==order
66                     report.test_lh > best_lh &&
67                         (best_lh=report.test_lh;best_rep=id)
68                     naive_lh=1. && (naive_lh=chain_reports[id].naive_lh)
69                     chain_reports[id].converged ?
70                         (push!(conv_statevec,id.K);

```

```

63         push!(conv_lh_vec,chain_reports[id].test_lh)) :
64             ↳ (push!(fail_statevec,id.K);
65             push!(fail_lh_vec,chain_reports[id].test_lh))
66     end
67
68     ↳ orddict[order]=Order_Report(conv_statevec,conv_lh_vec,fail_statevec,f
69 end
70 best_model=(best_rep,chain_reports[best_rep].final_hmm)
71 best_repset=[Chain_ID(partition, best_rep.K, best_rep.order, rep)
72     ↳ for rep in reps]
73
74     ↳ push!(reports,Partition_Report(partition,naive_lh,orddict,best_model,best_
75 end
76     return reports
77 end

```

15.2.17 /src/reports/replicate_convergence.jl

```

1 struct Replicate_Report
2     ids::Vector{Chain_ID} #replicates
3     state_vecs::Dict{Chain_ID,Matrix{Float64}} #vectors of autotransition
4         ↳ probabilities evolving by iterate, concatenated to matrices,
5         ↳ indexed by Chain_ID
6     emission_arrays::Dict{Chain_ID,Array{Float64}}
7         ↳ #iterate*symbol*state_vecs
8     sorted_id1_states::Vector{Integer}
9     sort_dicts::Dict{Chain_ID,Dict{Integer, Integer}}#emission channels
10        ↳ sorted on the basis of euclidean closeness to ids[1]
11     sorted_symbols::Dict{Integer,Vector{Integer}}
12
13     Replicate_Report(ids,state_vecs,emission_arrays,
14         ↳ sorted_id1_states,sort_dicts,sorted_symbols) =
15         ↳ assert_repreport(ids) &&
16         ↳ new(ids,state_vecs,emission_arrays,sorted_id1_states,sort_dicts,sorted_symbols)
17 end
18
19 function assert_repreport(ids::Vector{Chain_ID})
20     Ks=Vector{Int64}()
21     orders=Vector{Int64}()
22     replicates=Vector{Int64}()

```

```

17     obsids=Vector{String}()
18
19     for id in ids
20         push!(Ks,id.K)
21         push!(orders,id.order)
22         push!(obsids,id.obs_id)
23         push!(replicates, id.replicate)
24     end
25
26     length(unique(Ks)) > 1 && throw(ArgumentError("Replicate set includes
27     ↳ chains with different K state numbers! All chains must have HMMs
28     ↳ with the same number of states."))
29     length(unique(orders)) > 1 && throw(ArgumentError("Replicate set
30     ↳ includes chains with different order coding numbers! All chains
31     ↳ must have HMMs with the same order coding."))
32     !allunique(replicates) && throw(ArgumentError("Replicate set includes
33     ↳ chains with the same replicate #. Replicates should be unique!"))
34
35     return true
36   end
37
38   function
39     ↳ report_replicates(repset::Vector{Chain_ID},chains::Dict{Chain_ID,Vector{EM_step}})
40     assert_repreport(repset) #check that the repset will pass its
41     ↳ constructor before doing the work
42     emission_arrays=Dict{Chain_ID,AbstractArray{AbstractFloat}}()
43     state_arrays=Dict{Chain_ID,AbstractArray{AbstractFloat}}()
44
45     for id in repset
46       emission_array=zeros(length(chains[id]),4^(id.order+1),id.K)
47       state_array=zeros(length(chains[id]),id.K)
48       for (it, step) in enumerate(chains[id])
49         for k in 1:id.K
50           emission_array[it,:,k]=step.hmm.B[k].p
51           state_array[it,k]=step.hmm.A[k,k]
52         end
53       end
54       emission_arrays[id]=emission_array
55       state_arrays[id]=state_array
56     end
57
58     println(repset)
59     println(emission_arrays)

```

```

53    println(state_arrays)
54
55    ↵  sort_dicts,sorted_id1_states,sorted_symbols=sort_emitters_by_distance!(repset
56
57    return
58      ↵  Replicate_Report(repset,state_arrays,emission_arrays,sorted_id1_states,
59      ↵  sort_dicts,sorted_symbols)
60 end
61
62 function sort_emitters_by_distance!(repset,emission_arrays)
63   id1=repset[1]
64   length(repset)==1 && (return
65     ↵  Dict{Chain_ID,Dict{Integer,Integer}}(),[1:id1.K...],[1:4^id1.order+1])
66
67   final_emissions =
68     ↵  zeros(length(repset),size(emission_arrays[id1])[2:3]...)
69   for (n,id) in enumerate(repset)
70     final_emissions[n,:,:] = emission_arrays[id][end,:,:]
71   end
72
73   distances=zeros(length(repset)-1,id1.K,id1.K)
74   for rep in 1:length(repset)-1
75     for k in 1:id1.K, j in 1:id1.K
76
77       ↵  distances[rep,k,j]=euclidean(final_emissions[1,:,:k],final_emissions[1
78     end
79   end
80
81   sortdicts=Dict{Chain_ID,Dict{Integer,Integer}}()
82   sorted_id1_states=Vector{Integer}()
83   sorted_symbols=Dict{Integer,Vector{Integer}}()
84   for rep in 1:length(repset)-1
85     sortdicts[repset[rep+1]]=Dict{Integer,Integer}()
86   end
87
88   for i in 1:id1.K
89     id1_mindist_K=findmin([sum([minimum(distances[rep,k,:]) for rep
90       ↵  in 1:length(repset)-1]) for k in 1:id1.K])[2]#find the state
91     ↵  from the first replicate that has minimum cumulative distance
92     ↵  to the closest state in the other replicates, excluding
93     ↵  already-chosen states by spiking their values
94     push!(sorted_id1_states,id1_mindist_K)
95   for rep in 1:length(repset)-1

```

```

86         rep_k=findmin(distances[rep,id1_mindist_K,:])[2]
87         sortdicts[repset[rep+1]][id1_mindist_K]=rep_k
88         distances[rep,:,:rep_k] *= 1.0 #mask
89         distances[1,id1_mindist_K,:] *= 1.0 #mask
90     end
91
92
93     → symbol_distances = sum(hcat([euclidean.(final_emissions[1,:,:id1_mindist_K],
94     → for (n,id) in enumerate(repset[2:end])) ... ), dims=2)
95     sorted_symbols[id1_mindist_K]=sortperm(symbol_distances[:,1])
96 end
97
98
99
100 function Base.show(io::IO, report::Replicate_Report;
101   → top_states::Integer=report.ids[1].K, top_symbols::Integer=2)
102   printstyled(io, "REPLICATE CONVERGENCE REPORT\n", color=:green)
103   println(" -----")
104   → -----
105   → -----
106   for i in 1:top_states
107     id=report.ids[1]
108     id1_state=report.sorted_id1_states[i]
109     itmax=maximum([size(report.state_vecs[id],1) for id in
110       → report.ids])
111
112     autoplt=lineplot(report.state_vecs[id][:,id1_state],
113       title="Autotransition prob. convergence",
114       name=$(report.ids[1]) K$id1_state",
115       xlim=(0,ceil(itmax, sigdigits=2)),
116       ylim=(0,1),
117       xlabel="iterate",
118       ylabel="p")
119
120     → symplot=lineplot(report.emission_arrays[id][:,report.sorted_symbols[id1_st

```

```
121         ylim=(0,1),  
122         xlabel="Symbol 1 p",  
123         ylabel="S2 p")  
124  
125     for (n,id) in enumerate(report.ids)  
126         if n > 1  
127             lineplot!(autoplt,  
128                     → report.state_vecs[id][:,report.sort_dicts[id][id1_state]],  
129                     name="$(report.ids[n])"  
130                     → K$(report.sort_dicts[id][id1_state]))  
131             lineplot!(symplot,  
132                     → report.emission_arrays[id][:,report.sorted_symbols[id]]  
133                     → report.emission_arrays[id][:,report.sorted_symbols[id]]  
134                     name="$(report.ids[n])"  
135                     → K$(report.sort_dicts[id][id1_state]))  
136         end  
137     end  
138  
139         show(autoplt)  
140         println("\n")  
141         show(symplot)  
142         println("\n")  
143     end  
144 end
```

15.2.18 /src/utilities/BBG analysis.jl

```
1 #function to produce matrix of hmm chain parameter coordinate evolution-
  ↵ selects 3 matched emission parameters from 3+ state HMMs, one per
  ↵ state for 3 states. Matching is performed by minimizing euclidean
  ↵ distance between the state emission probability vectors B for the
  ↵ states to be matched (as EM-optimised replicates will have similar
  ↵ states organized in different orders)

2

3 function chain_3devo_coords(chains :: Vector{Vector{Any}})
  ↵ length(chains) ≤ 0 && throw(ArgumentError, "Argument must be vector of
  ↵ more than one hmm chain")
  ↵ coords=[Vector{Tuple{AbstractFloat,AbstractFloat,AbstractFloat}}]()
  ↵ for i in 1:length(chains)]
```

```

7   for step in chains[1]
8     hmm=step[2]
9     length(hmm.B)<3 && throw(ArgumentError, "3- or greater state hmms
10    ↪ required")
11    push!(coords[1], (hmm.B[1].p[1],hmm.B[2].p[1],hmm.B[3].p[1]))
12  end
13
14  for (c, chain) in enumerate(chains[2:end])
15    ref_idxs=Vector{Integer}()
16
17    ↪ ref_vecs=[chains[1][end][2].B[1].p,chains[1][end][2].B[2].p,chains[1][end]
18    end_hmm=chain[end][2]
19    length(end_hmm.B)<3 && throw(ArgumentError, "3- or greater state
20    ↪ hmms required")
21
22    for vec in ref_vecs
23      euclideans=Vector{AbstractFloat}()
24      for d in 1:length(end_hmm.B)
25        push!(euclideans, euclidean(vec, end_hmm.B[d].p))
26      end
27      min_euc_idx = findmin(euclideans)[2]
28      while min_euc_idx in ref_idxs
29        euclideans[min_euc_idx]=1.
30        min_euc_idx = findmin(euclideans)[2]
31      end
32      push!(ref_idxs,findmin(euclideans)[2])
33    end
34
35    for step in chain
36      hmm=step[2]
37      push!(coords[c+1],
38        ↪ (hmm.B[ref_idxs[1]].p[1],hmm.B[ref_idxs[2]].p[1],hmm.B[ref_idxs[3]].p
39      end
40    end
41
42    return coords
43 end

```

15.2.19 /src/utilities/BBG_progressmeter.jl

```

1 #UTILITY PROGRESSMETER, REPORTS WORKER NUMBER AND CURRENT ITERATE
2 mutable struct ProgressHMM{T<:Real} <: ProgressMeter.AbstractProgress

```

```

3   thresh::T
4   dt::AbstractFloat
5   val::T
6   counter::Integer
7   triggered::Bool
8   tfirst::AbstractFloat
9   tlast::AbstractFloat
10  printed::Bool      # true if we have issued at least one status
11    → update
12  desc::AbstractString # prefix to the percentage, e.g. "Computing ... "
13  color::Symbol        # default to green
14  output::IO           # output stream into which the progress is
15    → written
16  numprintedvalues::Integer # num values printed below progress in
17    → last iteration
18  offset::Integer       # position offset of progress bar
19    → (default is 0)
20  steptime::AbstractFloat
21
22  function ProgressHMM{T}(thresh;
23                            dt::Real=0.1,
24                            desc::AbstractString="Progress: ",
25                            color::Symbol=:green,
26                            output::IO=stderr,
27                            offset::Integer=0,
28                            start_it::Integer=1) where T
29
30    tfirst = tlast = time()
31    printed = false
32    new{T}(thresh, dt, typemax(T), start_it, false, tfirst, tlast,
33          → printed, desc, color, output, 0, offset, 0.0)
34  end
35 end
36
37 ProgressHMM(thresh::Real, dt::Real=0.1, desc::AbstractString="Progress:
38   → ",
39   → color::Symbol=:green, output::IO=stderr;
40   → offset::Integer=0, start_it::Integer=1) =
41   → ProgressHMM{typeof(thresh)}(thresh, dt=dt, desc=desc,
42   → color=color, output=output, offset=offset,
43   → start_it=start_it)
44
45 ProgressHMM(thresh::Real, desc::AbstractString, offset::Integer=0,
46   → start_it::Integer=1) = ProgressHMM{typeof(thresh)}(thresh, desc=desc,
47   → offset=offset, start_it=start_it)

```

```

35
36 function update!(p::ProgressHMM, val, steptime; options ... )
37     p.val = val
38     p.counter += 1
39     p.steptime = steptime
40     updateProgress!(p; options ... )
41 end
42
43 function updateProgress!(p::ProgressHMM; showvalues = Any[], valuecolor =
44     :blue, offset::Integer = p.offset, keep = (offset == 0))
45     p.offset = offset
46     t = time()
47     if p.val ≤ p.thresh && !p.triggered
48         p.triggered = true
49         if p.printed
50             p.triggered = true
51             dur = ProgressMeter.durationstring(t-p.tfirst)
52             msg = @sprintf "%s Time: %s (%d iterations)" p.desc dur
53             → p.counter
54             print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
55             ProgressMeter.move_cursor_up_while_clearing_lines(p.output,
56             → p.numprintedvalues)
57             ProgressMeter.printover(p.output, msg, p.color)
58             ProgressMeter.printvalues!(p, showvalues; color = valuecolor)
59             if keep
60                 println(p.output)
61             else
62                 print(p.output, "\r\u1b[A" ^ (p.offset +
63                 → p.numprintedvalues))
64             end
65         end
66         return
67     end
68     if t > p.tlast+p.dt && !p.triggered
69         elapsed_time = t - p.tfirst
70         msg = @sprintf "%s (convergence: %g ⇒ %g, iterate: %g, step
71             → time: %s)" p.desc p.val p.thresh p.counter hmss(p.steptime)
72         print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
73         ProgressMeter.move_cursor_up_while_clearing_lines(p.output,
74             → p.numprintedvalues)
75         ProgressMeter.printover(p.output, msg, p.color)
76         ProgressMeter.printvalues!(p, showvalues; color = valuecolor)
77     end

```

```

72     print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))
73     # Compensate for any overhead of printing. This can be
74     # especially important if you're running over a slow network
75     # connection.
76     p.tlast = t + 2*(time()-t)
77     p.printed = true
78   end
79 end
80
81   function hmss(dt)
82     isnan(dt) && return "NaN"
83     (h,r) = divrem(dt,60*60)
84     (m,r) = divrem(r, 60)
85     (isnan(h)||isnan(m)||isnan(r)) && return "NaN"
86     string(Int(h),":",Int(m),":",Int(ceil(r)))
87   end

```

15.2.20 /src/utilities/HMM_init.jl

```

1 function autotransition_init(K::Integer, order::Integer,
2   ← partition::String="")
3   a = rand(Dirichlet(ones(K)/K)) #uninformative prior on initial
4   ← state probabilities
5   A = strong_autotrans_matrix(K)
6   no_emission_symbols = Int(4^(order+1)) #alphabet size for the
7   ← order
8   emission_dists = [generate_emission_dist(no_emission_symbols) for
9   ← i in 1:K]
10  #generate the BHMM with the appropriate transition matrix and
11  ← emissions distributions
12  hmm = BHMM(a, A, emission_dists, partition)
13
14 end
15
16   #function to construct BHMM transition matrix with strong
17   ← priors on auto-transition
18   function strong_autotrans_matrix(states::Integer,
19   ← prior_dope::AbstractFloat=(states*250.0),
20   ← prior_background::AbstractFloat=.1)
21   transition_matrix=zeros(states,states)
22   for k in 1:states
23     dirichlet_params = fill(prior_background, states)
24     dirichlet_params[k] = prior_dope

```

```

15         transition_matrix[k,:] =
16             ↳ rand(Dirichlet(dirichlet_params))
17     end
18     return transition_matrix
19 end
20
21 #function to construct BHMM state emission distribution from
22     ↳ uninformative dirichlet over the alphabet size
23 function generate_emission_dist(no_emission_symbols,
24     ↳ prior=Dirichlet(ones(no_emission_symbols)/no_emission_symbols))
25     return Categorical(rand(prior))
26 end

```

15.2.21 /src/utilities/load_balancer.jl

```

1 struct LoadConfig
2     k_range::UnitRange
3     o_range::UnitRange
4     blacklist::Vector{Chain_ID}
5     whitelist::Vector{Chain_ID}
6     function LoadConfig(k_range, o_range; blacklist=Vector{Chain_ID}(),
7         ↳ whitelist=Vector{Chain_ID}())
8         assert_loadconfig(k_range, o_range) && new(k_range, o_range,
9             ↳ blacklist, whitelist)
10    end
11 end
12
13 function assert_loadconfig(k_range,o_range)
14     k_range[1] < 1 && throw(ArgumentError("Minimum value of LoadConfig
15         ↳ k_range is 1!"))
16     o_range[1] < 0 && throw(ArgumentError("Minimum value of LoadConfig
17         ↳ o_range is 0!"))
18     return true
19 end
20
21 #subfunc to handle balancing memory load on dissimilar machines in
22     ↳ cluster
23 function load_balancer(no_models::Integer, hmm_jobs::RemoteChannel,
24     ↳ config::LoadConfig, timeout::Integer=300)
25     jobid::Chain_ID, start_iterate::Integer, hmm::BHMM,
26     ↳ job_norm::AbstractFloat, observations::Matrix = take!(hmm_jobs)

```

```

21     starttime=time()
22     while (jobid.K < config.k_range[1] || jobid.K > config.k_range[end]
23         || jobid.order < config.o_range[1] || jobid.order >
24         config.o_range[end] || jobid in config.blacklist ||
25         (length(config.whitelist)>0 && !(jobid in config.whitelist))) &&
26         (time()-starttime ≤ timeout) #while a job prohibited by load
27         → table, keep putting the job back and drawing a new one
28         put!(hmm_jobs, (jobid, start_iterate, hmm, job_norm,
29             → observations))
30         jobid, start_iterate, hmm, job_norm, observations =
31             → take!(hmm_jobs)
32     end
33     (time()-starttime > timeout) ? (return 0,0,0,0,0) : (return jobid,
34         → start_iterate, hmm, job_norm, observations)
35 end

```

15.2.22 /src/utilities/log_prob_sum.jl

```

1      #subfuncs to handle sums of log probabilities that may
2          → include -Inf (ie p=0), returning -Inf in this case
3          → rather than NaNs
4      function lps(adjuvants)
5          prob = sum(adjuvants) ; isnan(prob) ? - Inf : prob
6      end
7
7      function lps(base, adjuvants ...)
8          prob = base+sum(adjuvants) ; isnan(prob) ? -Inf :
9              → prob
10     end

```

15.2.23 /src/utilities/model_display.jl

```

1 cs_dict=Dict('A'=>:green, 'C'=>:blue, 'G'=>:yellow, 'T'=>:red)
2
3 function print_emitters(state::Categorical) #print informative symbols
4     → from the state's emission distribution, if any
5     order=Integer(log(4,length(state.p))-1)
6     alphabet=CompoundAlphabet(ACGT,order)
7     bits=log(2,length(state.p)) #higher order hmms express more
8         → information per symbol

```

```

7     dummy_p=state.p.+10^-99 #prevent -Inf or NaN from zero probability
   ↵ symbols
8     infoscore=(bits+sum(x*log(2,x) for x in dummy_p))
9     infovec=[x*infoscore for x in dummy_p]
10    infovec./=bits
11    print("<<< ")
12    for (symbol,symbol_prob) in enumerate(infovec)
13        if symbol_prob ≥ .05
14            str=string(alphabet.integers[symbol])
15            if symbol_prob ≥ .7
16                for char in str
17                    printstyled(stdout, char; bold=true,
   ↵ color=cs_dict[char])
18            end
19            elseif .7 > symbol_prob ≥ .25
20                for char in str
21                    printstyled(stdout, char; color=cs_dict[char])
22                end
23            elseif .25 > symbol_prob
24                for char in str
25                    printstyled(stdout, lowercase(char);
   ↵ color=cs_dict[char])
26                end
27            end
28            print(" ")
29        end
30    end
31    println(">>>")
32 end

```

15.2.24 /src/utilities/observation_coding.jl

```

1 #KMER ORDER/SEQUENCE INTEGER CODING UTILITIES
2 #higher order DNA alphabet
3 struct CompoundAlphabet
4     symbols::Dict{Mer{DNAAlphabet{2}}, Integer}
5     integers::Dict{Integer,Mer{DNAAlphabet{2}}}
6     #build a CompoundAlphabet for DNA of some order_no
7     function CompoundAlphabet(alphabet::Tuple, order_no::Integer)
8         symbols = Dict{Mer{DNAAlphabet{2}}, Integer}()
9         integers = Dict{Integer,Mer{DNAAlphabet{2}}}()
10

```

```

11     tuples = Array{Tuple}
12     if order_no > 0
13         tuples = collect(Iterators.product([alphabet for order in
14             ← 0:order_no] ... ))
15     else #0th order compound product of an alphabet is that alphabet
16         tuples = collect(Iterators.product(alphabet))
17     end
18
19     @inbounds for index in eachindex(tuples)
20         tuple_seq = Mer{DNAAlphabet{2}}(collect(tuples[index]))
21         integers[index] = tuple_seq
22         symbols[tuple_seq] = index
23     end
24
25     new(symbols,integers)
26 end
27
28
29 struct N_Order_ntSequence
30     alphabet::CompoundAlphabet
31     seq_lengths::Vector{Integer}
32     order_kmers::Vector{Vector{Mer{DNAAlphabet{2}}}}
33 end
34
35
36 function code_job_obs(job_ids::Vector{Chain_ID},
37     ← obs_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}})
38     code_jobs=Vector{Tuple{String,Integer}}(){}
39     for id in job_ids #assemble a vector of observation encoding jobs
40         code_job=(id.obs_id, id.order)
41         !(code_job in code_jobs) && push!(code_jobs, code_job)
42     end
43
44     code_dict = Dict{Tuple{String,Integer}, AbstractArray}(){}
45     @showprogress 1 "Encoding observations ..." for (obs_id, order) in
46         ← code_jobs #build the appropriate sample sets once
47         order_seqs = get_order_n_seqs(obs_sets[obs_id],order) #get the
48         ← kmer sequences at the appropriate order
49         coded_seqs = code_seqs(order_seqs, sorted=true) #numerically code
49         ← the sequences in trainable format
50         code_dict[(obs_id, order)] = coded_seqs
51     end

```

```

49     return code_dict
50 end
51
52 #from a vector of LongSequences, get
53 function get_order_n_seqs(seqs::Vector{LongSequence{DNAAlphabet{2}}}),
54     ↳ order_no::Integer, base_tuple::Tuple=ACGT)
55     kmer_vecs = Vector{Vector{Mer{DNAAlphabet{2}}}}()
56     length_vec = Vector{Integer}()
57     window = order_no + 1
58
59     for seq in seqs
60         kmer_vec = Vector{Mer{DNAAlphabet{2}}}()
61         @inbounds for (i, kmer) in
62             ↳ collect(each(Mer{DNAAlphabet{2}},window),seq))
63             push!(kmer_vec, kmer)
64         end
65
66         push!(kmer_vecs, kmer_vec)
67         push!(length_vec, length(seq))
68     end
69
70     return NordSeqs = N_Order_ntSequence(CompoundAlphabet(base_tuple,
71         ↳ order_no), length_vec, kmer_vecs)
72 end
73
74 #convert tuple kmers to symbol codes
75 function code_seqs(input::N_Order_ntSequence, offsets::Vector=[0 for i in
76     ↳ 1:length(input.order_kmers)]; sorted::Bool=false)
77     symbol_no=length(input.alphabet.symbols)
78     if symbol_no ≤ typemax(UInt8)
79         integer_type = UInt8
80     elseif typemax(UInt8) < symbol_no < typemax(UInt16)
81         integer_type = UInt16
82     else
83         integer_type = UInt32
84     end
85
86     alphabet = input.alphabet
87     output = zeros(integer_type, length(input.order_kmers),
88         ↳ (maximum([length(seq) for seq in input.order_kmers])+1)) #leave 1
89         ↳ missing value after the longest sequence for indexing sequence
90         ↳ length in CLHMM messages
91     sorted && (sort_idxs = sortperm(input.seq_lengths,rev=true))

```

```

85     sorted ? input_seqs = input.order_kmers[sort_idxs] : input_seqs =
86             ↵ input.order_kmers
87
88     for (i, seq) in enumerate(input_seqs)
89         for t in 1:length(seq)
90             curr_kmer = seq[t]
91             curr_code = alphabet.symbols[curr_kmer]
92             output[i,t+offsets[i]]=curr_code
93         end
94     end
95     return output
96 end

```

15.2.25 /src/utilities/utilities.jl

```

1 """
2 Utility functions for learning and using background genomic hidden markov
3     ↵ models
4 """
5
6 #function to split random sample dataframe into training and test sets
7     ↵ (divide total sequence length by half)
8 function split_obs_sets(sample_dfs::Dict{String,DataFrame})
9     training_sets = Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}()
10    test_sets = Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}()
11
12    for (partition_id, partition) in sample_dfs
13        partition = partition[shuffle(1:size(partition, 1)),:] #shuffle
14            ↵ the samples to avoid any effect from sampling without
15            ↵ replacement
16        partition.sampleLength = (partition.SampleEnd -
17            ↵ partition.SampleStart) .+ 1
18        midway = sum(partition.sampleLength)÷2
19        split_index = 0
20        counter = 0
21        while split_index == 0
22            ↵ counter += 1
23            length_sum = sum(partition.sampleLength[1:counter])
24            if length_sum > midway
25                split_index = counter
26            end
27        end
28    end
29
30 end

```

```

23     training_sets[partition_id] =
24         ↳ partition.SampleSequence[1:split_index-1]
25     test_sets[partition_id] =
26         ↳ partition.SampleSequence[split_index:end]
27   end
28   return training_sets, test_sets
29 end

```

15.2.26 /test/ref_fns.jl

```

1 #slow algo written in mimicry of Churbanov & Winters
2 function old_linear(hmm, observations, obs_lengths)
3     O = size(observations)[2]
4     a = log.(hmm.A); a = log.(hmm.a)
5     N = length(hmm.B); B = length(hmm.B[1].support); b =
6         ↳ [log(hmm.B[m].p[y]) for m in 1:N, y in 1:B]
7     α1oi = zeros(O,N); β1oi = zeros(O,N); Eoi = zeros(O,N,B); Toij =
8         ↳ zeros(O,N,N); Aoi = zeros(O,N); log_pobs=zeros(O); γt=0
9
10    for o in 1:O
11        #INITIALIZATION
12        T = obs_lengths[o]; βoi_T = zeros(N) #log betas at T initialised
13        ↳ as zeros
14        EiT = fill(-Inf,B,N,N); TijT = fill(-Inf,N,N,N) #Ti,j(T,m) = 0
15        ↳ for all m; in logspace
16
17        @inbounds for m in 1:N, i in 1:N, y in 1:B
18            observations[T, o] = y && m == i ? (EiT[y, i, m] = 0) :
19                (EiT[y, i, m] = -Inf) #log Ei initialisation
20        end
21
22        #RECURRANCE
23        @inbounds for t in T-1:-1:1
24            βoi_t = similar(βoi_T); Tijt = similar(TijT); Eit =
25                ↳ similar(EiT)
26            Γ = observations[t+1,o]; γt = observations[t,o]
27            Γ=0 && println("hooo !! $o $t")
28            for m in 1:N
29                βoi_t[m] = logsumexp([lps(a[m,j], b[j,Γ], βoi_T[j]) for j
30                    ↳ in 1:N])
31                for i in 1:N
32                    for j in 1:N
33                        Tijt[i,j] = exp(Eit[y, i, m])
34                    end
35                end
36            end
37        end
38    end
39 end

```

```

27         Tijt[i, j, m] = logsumexp([lps(a[m,n], TijT[i, j,
28             ↵ n], b[n,Γ]) for n in 1:N])
29         i==m && (Tijt[i, j, m] = logaddexp(Tijt[i, j, m],
30             ↵ lps(βoi_T[j], a[m,j], b[j, Γ])))
31     end
32     for y in 1:B
33         Eit[y, i, m] = logsumexp([lps(b[n,Γ], a[m,n],
34             ↵ EiT[y, i, n]) for n in 1:N])
35         i==m && y==yt && (Eit[y, i, m] = logaddexp(Eit[y,
36             ↵ i, m], βoi_t[m])))
37     end
38     end
39
40     βoi_T = βoi_t; TijT = Tijt; EiT = Eit
41 end
42
43 #TERMINATION
44
45     Γ = observations[1,o]
46     α1oi[o,:]= [lps(a[i], b[i, Γ]) for i in 1:N]
47     β1oi[o,:]= βoi_T
48     log_pobs[o]= logsumexp(lps.(α1oi[o,:], βoi_T[:]))
49     Eoi[o,:,:]= [logsumexp([lps(EiT[y,i,m], a[m], b[m,yt]) for m in
50             ↵ 1:N]) for i in 1:N, y in 1:B]
51     Toij[o,:,:,:]= [logsumexp([lps(TijT[i,j,m], a[m], b[m,yt]) for m
52             ↵ in 1:N]) for i in 1:N, j in 1:N]
53 end
54
55 #INTEGRATE ACROSS OBSERVATIONS AND SOLVE FOR NEW BHMM PARAMS
56
57     obs_penalty=log(0)
58     a_o=α1oi.+β1oi.-logsumexp.(eachrow(α1oi.+β1oi))
59     new_a=logsumexp.(eachcol(a_o))-obs_penalty
60
61     a_int = Toij.-logsumexp.([Toij[o,i,:] for o in 1:O, i in 1:N])
62     new_a = logsumexp.([a_int[:,i,j] for i in 1:N, j in
63             ↵ 1:N])-obs_penalty
64
65     e_int=Eoi.-logsumexp.([Eoi[o,j,:] for o in 1:O, j in 1:N])
66     new_b=logsumexp.([e_int[:,j,d] for d in 1:B, j in 1:N])-obs_penalty
67     new_D=[Categorical(exp.(new_b[:,i])) for i in 1:N]
68
69
70     return typeof(hmm)(exp.(new_a), exp.(new_a), new_D), lps(log_pobs)
71 end

```

```

63
64 #HMMBase
65 function mouchet_mle_step(hmm::BHMM, observations) where F
66     # NOTE: This function works but there is room for improvement.
67
68     log_likelihoods = mouchet_log_likelihoods(hmm, observations)
69
70     log_α = mouchet_messages_forwards_log(hmm.a, hmm.A, log_likelihoods)
71     log_β = mouchet_messages_backwards_log(hmm.A, log_likelihoods)
72     log_A = log.(hmm.A)
73
74     normalizer = logsumexp(log_α[1,:] + log_β[1,:])
75
76     # E-step
77
78     T, K = size(log_likelihoods)
79     log_ξ = zeros(T, K, K)
80
81     @inbounds for t = 1:T-1, i = 1:K, j = 1:K
82         log_ξ[t,i,j] = log_α[t,i] + log_A[i,j] + log_β[t+1,j] +
83             ↳ log_likelihoods[t+1,j] - normalizer
84     end
85
86     ξ = exp.(log_ξ)
87     ξ ./= sum(ξ, dims=[2,3])
88
89     # M-step
90     new_A = sum(ξ[1:end-1,:,:], dims=1)[1,:,:] #index fix-MM
91     new_A ./= sum(new_A, dims=2)
92
93     new_a = exp.((log_α[1,:] + log_β[1,:]) .- normalizer)
94     new_a ./= sum(new_a)
95
96     # TODO: Cleanup/optimize this part
97     γ = exp.((log_α .+ log_β) .- normalizer)
98
99     B = Categorical[]
100    for (i, d) in enumerate(hmm.B)
101        # Super hacky ...
102        # https://github.com/JuliaStats/Distributions.jl/issues/809
103        push!(B, fit_mle(Categorical, permutedims(observations), γ[:,i]))
104    end

```

```
105     typeof(hmm)(new_a, new_A, B), normalizer
106 end
107
108
109 function mouchet_log_likelihoods(hmm, observations)
110     hcat(map(d → logpdf.(d, observations), hmm.B) ... )
111 end
112
113
114
115 function mouchet_messages_forwards_log(init_distn, trans_matrix,
116   ↪ log_likelihoods)
116   # OPTIMIZE
117   log_alphas = zeros(size(log_likelihoods))
118   log_trans_matrix = log.(trans_matrix)
119   log_alphas[1,:] = log.(init_distn) .+ log_likelihoods[1,:]
120   @inbounds for t = 2:size(log_alphas)[1]
121       for i in 1:size(log_alphas)[2]
122           log_alphas[t,i] = logsumexp(log_alphas[t-1,:] .+
123             ↪ log_trans_matrix[:,i]) + log_likelihoods[t,i]
124       end
125   end
126   log_alphas
126 end
127
128 function mouchet_messages_backwards_log(trans_matrix, log_likelihoods)
129   # OPTIMIZE
130   log_betas = zeros(size(log_likelihoods))
131   log_trans_matrix = log.(trans_matrix)
132   @inbounds for t = size(log_betas)[1]-1:-1:1
133       tmp = view(log_betas, t+1, :) .+ view(log_likelihoods, t+1, :)
134       @inbounds for i in 1:size(log_betas)[2]
135           log_betas[t,i] = logsumexp(view(log_trans_matrix, i, :) .+
136             ↪ tmp)
137       end
138   end
139   log_betas
139 end
```

15.2.27 /test/runtests.jl

```

1 using
  ↵ BioBackgroundModels,BioSequences,DataFrames,Distributed,Distributions,FASTX,Progr
2 import BioBackgroundModels: autotransition_init, load_balancer,
  ↵ CompoundAlphabet, get_order_n_seqs, code_seqs, code_job_obs,
  ↵ make_padded_df, add_partition_masks!, partition_genome_coordinates,
  ↵ divide_partitions_by_scaffold, mask_sequence_by_partition,
  ↵ find_position_partition, setup_sample_jobs, rectify_identifier,
  ↵ add_metacoordinates!, meta_to_feature_coord, get_feature_row_index,
  ↵ get_strand_dict, build_scaffold_seq_dict,
  ↵ get_feature_params_from_metacoord, determine_sample_window,
  ↵ fetch_sequence, get_sample_set, get_sample, assert_hmm, linear_step,
  ↵ get_BGHMM_symbol_lh,make_padded_df,add_partition_masks!,BGHMM_likelihood_calc,lin
  ↵ t_add_categorical_counts!
3 import Serialization: deserialize
4 include("synthetic_sequence_gen.jl")
5 include("ref_fns.jl")
6
7 #JOB FILEPATHS
8 #GFF3 feature database, FASTA genome and index paths
9 Sys.islinux() ? genome = (@__DIR__) * "/synthetic.fna" : genome =
  ↵ (@__DIR__) * "\\synthetic.fna"
10 Sys.islinux() ? index = (@__DIR__) * "/synthetic.fna.fai" : index =
  ↵ (@__DIR__) * "\\synthetic.fna.fai"
11 Sys.islinux() ? gff = (@__DIR__) * "/synthetic.gff3" : gff = (@__DIR__)
  ↵ * "\\synthetic.gff3"
12 Sys.islinux() ? posfasta = (@__DIR__) * "/syntheticpos.fa" : posfasta =
  ↵ (@__DIR__) * "\\syntheticpos.fa"
13 !isfile(genome) && print_synthetic_fasta(genome)
15 !isfile(index) && print_synthetic_index(index)
16 !isfile(gff) && print_synthetic_gff(gff)
17 !isfile(posfasta) && print_synthetic_position(posfasta)
18
19 Random.seed!(1)
20
21 @testset "BHMM/BHMM.jl constructors" begin
22     good_a = [.20,.30,.20,.30]
23     bad_probvec_a = [.25,.25,.25,.27]
24     good_A = fill(.25,4,4)
25     bad_probvec_A = deepcopy(good_A)
26     bad_probvec_A[:,3] *= .27

```

```

27     bad_size_A = fill(.25,5,4)
28     good_D = Categorical([.25,.25,.25,.25])
29     bad_size_D = Categorical([.2,.2,.2,.2,.2])
30     good_Dvec = [good_D for i in 1:4]
31     bad_size_Dvec =[good_D, good_D, good_D, bad_size_D]
32     bad_length_Dvec=[good_D for i in 1:5]

33
34     @test_throws ArgumentError BHMM(bad_probvec_a, good_A, good_Dvec)
35     @test_throws ArgumentError BHMM(good_a, bad_probvec_A, good_Dvec)
36     @test_throws ArgumentError BHMM(good_a, bad_size_A, good_Dvec)
37     @test_throws ArgumentError BHMM(good_a, good_A, bad_size_Dvec)
38     @test_throws ArgumentError BHMM(good_a, good_A, bad_length_Dvec)

39
40     hmm=BHMM(good_a, good_A, good_Dvec)
41     @test typeof(hmm)==BHMM{Float64}
42     @test hmm.a==good_a
43     @test hmm.A==good_A
44     @test hmm.B==good_Dvec
45     @test size(hmm) == (4,4)

46
47     hmm_copy=copy(hmm)
48     @test hmm.a==hmm_copy.a
49     @test hmm.A==hmm_copy.A
50     @test hmm.B==hmm_copy.B

51
52     hmm2=BHMM(good_A, good_Dvec)
53     @test hmm2.a==[.25,.25,.25,.25]
54     @test hmm2.A==good_A
55     @test hmm2.B==good_Dvec

56 end

57
58 @testset "BHMM/chain.jl Chain_ID and EM_step constructors" begin
59     obs_id="test"
60     K,zero_k,neg_K=4,0,-1
61     order,badorder=0,-1
62     replicate,zero_rep,neg_rep=1,0,-1

63
64     @test_throws ArgumentError Chain_ID(obs_id, zero_k, order, replicate)
65     @test_throws ArgumentError Chain_ID(obs_id, neg_K, order, replicate)
66     @test_throws ArgumentError Chain_ID(obs_id, K, badorder, replicate)
67     @test_throws ArgumentError Chain_ID(obs_id, K, order, zero_rep)
68     @test_throws ArgumentError Chain_ID(obs_id, K, order, neg_rep)

69

```

```

70     id=Chain_ID(obs_id, K, order, replicate)
71     @test typeof(id)=Chain_ID
72     @test id.obs_id==obs_id
73     @test id.K==K
74     @test id.order==order
75     @test id.replicate==replicate
76
77     good_A = fill(.25,4,4)
78     good_D = Categorical([.25,.25,.25,.25])
79     good_Dvec = [good_D for i in 1:4]
80     hmm=BHMM(good_A,good_Dvec)
81     log_p=-.001
82
83     @test_throws ArgumentError EM_step(0,hmm,0.0,0.0,true)
84     @test_throws ArgumentError EM_step(-1,hmm,0.0,0.0,true)
85     @test_throws ArgumentError EM_step(1,hmm,1.0,0.0,true)
86
87     steppe=EM_step(1,hmm,log_p,0.0,false)
88     @test typeof(steppe)=EM_step
89     @test steppe.iterate==1
90     @test steppe.hmm==hmm
91     @test steppe.log_p==log_p
92     @test steppe.delta==0.0
93     @test steppe.converged==false
94 end
95
96 @testset "utilities/log_prob_sum.jl Log probability summation functions" begin
97     @test isapprox(lps([.1, .1, .1]), .3)
98     @test lps([-Inf, .1, .1]) == -Inf
99     @test isapprox(lps(.1, .1, .1), .3)
100    @test lps(-Inf, .1, .1) == -Inf
101    @test lps(.1, -Inf, .1) == -Inf
102 end
103
104 @testset "utilities/HMM_init.jl initialization functions" begin
105     K=4
106     order=2
107     hmm=autotransition_init(K,order)
108     @test typeof(hmm)==BHMM{Float64}
109     @test size(hmm)==(4,64)
110 end
111

```

```

112 @testset "utilities/load_balancer.jl EM_converge load balancing structs
113   ↪ and functions" begin
114     @test_throws ArgumentError LoadConfig(0:5,1:5)
115     @test_throws ArgumentError LoadConfig(1:5,-1:5)
116
117     ↪ job_ids=[Chain_ID("test",6,2,1),Chain_ID("test",4,2,1),Chain_ID("test",2,0,1)]
118
119     ↪ obs_sets=Dict("test"⇒[LongSequence{DNAAlphabet{2}}]("AAAAAAAAAAAAAA"))
120
121     K_exclusive_config=LoadConfig(1:1,0:0)
122     O_exclusive_config=LoadConfig(1:6,3:4)
123     blacklist_config=LoadConfig(1:6,0:2,blacklist=job_ids)
124
125     ↪ whitelist_config=LoadConfig(1:6,0:2,whitelist=[Chain_ID("test",6,2,2)])
126     high_config=LoadConfig(4:6,0:2,whitelist=job_ids)
127
128     no_input_hmms, chains, input_channel, output_channel =
129       ↪ setup_EM_jobs!(job_ids,obs_sets)
130     @test load_balancer(no_input_hmms, input_channel, K_exclusive_config,
131       ↪ 3) = (0,0,0,0,0)
132
133     no_input_hmms, chains, input_channel, output_channel =
134       ↪ setup_EM_jobs!(job_ids,obs_sets)
135     @test load_balancer(no_input_hmms, input_channel, O_exclusive_config,
136       ↪ 3) = (0,0,0,0,0)
137
138     no_input_hmms, chains, input_channel, output_channel =
139       ↪ setup_EM_jobs!(job_ids,obs_sets)
140     high_set=load_balancer(no_input_hmms, input_channel, high_config)
141
142     @test high_set[1]=Chain_ID("test",6,2,1)
143     @test high_set[2]=1

```

```

142 @test typeof(high_set[3])=BHMM{Float64}
143 @test
144     ↪ high_set[4]=obs_lh_given_hmm(high_set[5],high_set[3],linear=false)
145 @test typeof(high_set[5])=Matrix{UInt8}
146 end
147
148 @testset "utilities/observation_coding.jl Order coding structs and
149     ↪ functions" begin
150     compound_alphabet=CompoundAlphabet(ACGT, 2)
151     @test
152         ↪ typeof(compound_alphabet.symbols)=Dict{Mer{DNAAlphabet{2}},Integer}
153     @test length(compound_alphabet.symbols)=64
154     @test compound_alphabet.symbols[Mer{DNAAlphabet{2}}("AAA")]=1
155     @test compound_alphabet.symbols[Mer{DNAAlphabet{2}}("TTT")]=64
156
157     test_seqs =
158         ↪ [BioSequences.LongSequence{DNAAlphabet{2}}("ACGTACGTACGTACGT"),BioSequences.L
159     target0= [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 0; 4 4 4 4 4 4 4 4 0 0 0 0 0
160         ↪ 0 0 0 0 0]
161     target2= [37 58 15 20 37 58 15 20 37 58 15 20 37 58 0; 64 64 64 64 64
162         ↪ 0 0 0 0 0 0 0 0 0]
163
164     order0_seqs = get_order_n_seqs(test_seqs,0)
165     @test typeof(order0_seqs)=BioBackgroundModels.N_Order_ntSequence
166     for (mer, code) in CompoundAlphabet(ACGT,0).symbols
167         @test order0_seqs.alphabet.symbols[mer]==code
168     end
169     @test order0_seqs.seq_lengths==[16,7]
170     @test order0_seqs.order_kmers[1][1]==Mer{DNAAlphabet{2}}("A")
171     code0_seqs = code_seqs(order0_seqs)
172     @test target0 == code0_seqs
173
174     order2_seqs = get_order_n_seqs(test_seqs,2)
175     @test typeof(order2_seqs)=BioBackgroundModels.N_Order_ntSequence
176     for (mer, code) in CompoundAlphabet(ACGT,2).symbols
177         @test order2_seqs.alphabet.symbols[mer]==code
178     end
179     @test order2_seqs.seq_lengths==[16,7]
180     @test order2_seqs.order_kmers[1][1]==Mer{DNAAlphabet{2}}("ACG")
181     code2_seqs = code_seqs(order2_seqs)
182     @test target2 == code2_seqs
183
184     bw_o0_seqs = get_order_n_seqs(test_seqs[end:-1:1], 0)

```

```

179     sorted_code_seqs = code_seqs(bw_o0_seqs, sorted=true)
180     @test target0 == sorted_code_seqs
181
182     obs_sets=Dict("test1"⇒test_seqs,
183                   "test2"⇒test_seqs)
184
185     → job_ids=[Chain_ID("test1",4,0,1),Chain_ID("test2",4,0,1),Chain_ID("test1",2,2)
186
187     code_dict=code_job_obs(job_ids,obs_sets)
188     @test ("test1",0) in keys(code_dict)
189     @test ("test1",2) in keys(code_dict)
190     @test ("test2",0) in keys(code_dict)
191     @test !(("test2",2) in keys(code_dict))
192     @test code_dict[("test1",0)]==target0
193     @test code_dict[("test1",2)]==target2
194     @test code_dict[("test2",0)]==target0
195
196     #test type selection for high order numbers
197     o5_nos=get_order_n_seqs(test_seqs,5)
198     o5_codes=code_seqs(o5_nos)
199     @test typeof(o5_codes)==Matrix{UInt16}
200
201     o7_nos=get_order_n_seqs(test_seqs,7)
202     o7_codes=code_seqs(o7_nos)
203     @test typeof(o7_codes)==Matrix{UInt32}
204
205 end
206
207 @testset "genome_sampling/partition_masker.jl functions" begin
208     synthetic_seq =
209         → BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())
210     position_start = 501
211     position_length=141
212     position_pad=350
213     perigenic_pad = 250
214
215     position_df = make_padded_df(posfasta, gff, genome, index,
216         → position_pad)
217     add_partition_masks!(position_df, gff, perigenic_pad)
218
219     @test position_df.SeqID[1]=="1"

```

```

217  @test length(position_df.PadSeq[1]) == position_pad+position_length
218  ↳   ==
219  ↳   length(position_df.PadStart[1]:position_df.PadStart[1]+position_length+positi
220  ↳   = position_df.End[1]-position_df.Start[1]+1+position_pad =
221  ↳   size(position_df.MaskMatrix[1])[1]
222  @test
223  ↳   synthetic_seq[position_df.PadStart[1]:position_df.End[1]]==position_df.PadSeq
224
225  mm = position_df.MaskMatrix[1]
226  idx=1#pos 151
227  @test sum(mm[idx:idx+99,1] .== 1)==length(mm[idx:idx+99,1])
228  idx+=100#pos 251
229  @test sum(mm[idx:idx+259,1] .== 2)==length(mm[idx:idx+259,1])
230  idx+=260 #pos 511
231  @test sum(mm[idx:idx+59,1] .== 3)==length(mm[idx:idx+59,1])
232  idx+=60#pos 571
233  @test sum(mm[idx:idx+29,1] .== 2)==length(mm[idx:idx+29,1])
234  idx+=30#pos601
235  @test sum(mm[idx:idx+40,1] .== 3)==length(mm[idx:idx+40,1])
236
237  #test exception
238  position_df = make_padded_df(posfasta, gff, genome, index,
239  ↳   position_pad)
240
241  partition_coords_dict=partition_genome_coordinates(gff,
242  ↳   perigenic_pad)
243
244  ↳   partitioned_scaffolds=divide_partitions_by_scaffold(partition_coords_dict)
245  scaffold_coords_dict = Dict{String,DataFrame}()
246  scaffold="1"
247  for ((partition, part_scaffold), df) in partitioned_scaffolds
248      if scaffold == part_scaffold
249          scaffold_coords_dict[partition] = df
250      end
251  end
252
253  @test_throws DomainError
254  ↳   mask_sequence_by_partition(1001,151,scaffold_coords_dict)
255 end
256
257 @testset "genome_sampling/sequence_sampler.jl functions &
258 ↳   API/genome_sampling.jl interface" begin
259     #rectifier tests

```

```
250 @test rectify_identifier("1")=="1"
251
252 partitions = 3 #exonic, periexonic, intragenic
253 sample_set_length=100
254 min_sample_window=5
255 max_sample_window=25
256 perigenic_pad=250
257 syn_intergenic_starts = [1]
258 syn_intergenic_ends = [250]
259 syn_periexonic_starts = [251,571,661,761,861,961]
260 syn_periexonic_ends = [510,600,700,800,900,1000]
261 syn_periexonic_strands = ['-', '-', '-', '-', '+', '-']
262 syn_exonic_starts = [511,601,701,801,901]
263 syn_exonic_ends = [570,660,760,860,960]
264 syn_exonic_strands = ['+', '+', '+', '+', '+', '+']
265 partition_lengths =
266   ↳ Dict("exon"⇒5*60,"intergenic"⇒250,"periexonic"⇒450)
267
268 # "Testing sequence sampler fns ... "
269 synthetic_seq =
270   ↳ BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())
271
272 # "Partitioning synthetic genome coordinates ... "
273 coordinate_partitions = partition_genome_coordinates(gff,
274   ↳ perigenic_pad)
275
276 @test coordinate_partitions["intergenic"].Start =
277   ↳ syn_intergenic_starts
278 @test coordinate_partitions["intergenic"].End = syn_intergenic_ends
279
280 @test coordinate_partitions["periexonic"].Start =
281   ↳ syn_periexonic_starts
282 @test coordinate_partitions["periexonic"].End = syn_periexonic_ends
283 @test coordinate_partitions["periexonic"].Strand =
284   ↳ syn_periexonic_strands
285
286 @test coordinate_partitions["exon"].Start = syn_exonic_starts
287 @test coordinate_partitions["exon"].End = syn_exonic_ends
288 @test coordinate_partitions["exon"].Strand = syn_exonic_strands
289
290 # "Checking sampling functions at all synthetic indices ... "
```

```

286     input_test_channel, completed_test_channel, progress_channel,
287     ↵ setlength = setup_sample_jobs(genome, index, gff,
288     ↵ sample_set_length, min_sample_window, max_sample_window,
289     ↵ perigenic_pad; deterministic=true)
290     while isready(input_test_channel)
291         genome_path, genome_index_path, partition_df, partitionid,
292         ↵ sample_set_length, sample_window_min, sample_window_max,
293         ↵ deterministic = take!(input_test_channel)
294
295         for feature in eachrow(partition_df)
296             seqid, scaffold_start, scaffold_end =
297                 ↵ meta_to_feature_coord(feature.MetaStart, feature.MetaEnd,
298                 ↵ partition_df)
299             @test scaffold_start == feature.Start && scaffold_end ==
300                 ↵ feature.End
301         end
302
303         stranded = get_strand_dict()[partitionid]
304         @test typeof(stranded) == Bool
305
306         scaffold_sequence_record_dict = build_scaffold_seq_dict(genome,
307             ↵ index)
308         @test scaffold_sequence_record_dict["1"] == synthetic_seq
309
310         partition_length = partition_df.MetaEnd[end]
311         @test partition_length == partition_lengths[partitionid]
312
313         metacoordinate_bitarray = trues(partition_df.MetaEnd[end])
314
315         for bitindex in findall(metacoordinate_bitarray)
316             feature_metaStart, feature_metaEnd, strand =
317                 ↵ get_feature_params_from_metacoord(bitindex, partition_df,
318                 ↵ stranded)
319             @test 1 ≤ feature_metaStart < feature_metaEnd ≤
320                 ↵ length(metacoordinate_bitarray)
321             @test feature_metaStart in partition_df.MetaStart
322             @test feature_metaEnd in partition_df.MetaEnd
323             if partitionid == "periexonic"
324                 @test strand == '-'
325             elseif partitionid == "exon"
326                 @test strand in ['-','+']
327             end
328             feature_length = length(feature_metaStart:feature_metaEnd)

```

```

317         window = determine_sample_window(feature_metaStart,
318                                         → feature_metaEnd, bitindex, metacoordinate_bitarray,
319                                         → sample_window_min, sample_window_max) #get an appropriate
320                                         → sampling window around the selected index, given the
321                                         → feature boundaries and params
322
323     @test 1 ≤ feature_metaStart ≤ window[1] <
324         → window[1]+sample_window_min-1 <
325         → window[1]+sample_window_max-1 ≤ window[2] ≤
326         → feature_metaEnd ≤ length(metacoordinate_bitarray)
327
328     sample_scaffoldid, sample_scaffold_start, sample_scaffold_end
329         → = meta_to_feature_coord(window[1],window[2],partition_df)
330
331     @test sample_scaffoldid = "1"
332
333     @test 1 ≤ sample_scaffold_start ≤
334         → sample_scaffold_start+sample_window_min ≤
335         → sample_scaffold_end ≤
336         → min(sample_scaffold_start+sample_window_max,1000) ≤ 1000
337
338
339     strand = '-' ?
340         → target_seq=reverse_complement(synthetic_seq[sample_scaffold_start:sample_scaffold_end])
341         → :
342
343     target_seq =
344         → synthetic_seq[sample_scaffold_start:sample_scaffold_end]
345
346
347     proposal_sequence = fetch_sequence(sample_scaffoldid,
348                                         → scaffold_sequence_record_dict, sample_scaffold_start,
349                                         → sample_scaffold_end, strand; deterministic=deterministic)
350                                         → #get the sequence associated with the sample window
351
352
353     @test proposal_sequence == target_seq
354
355 end
356
357
358 # "Verifying sampling channels ... "
359
360
361 input_sample_channel, completed_sample_channel, progress_channel,
362     → setlength = setup_sample_jobs(genome, index, gff,
363     → sample_set_length, min_sample_window, max_sample_window,
364     → perigenic_pad; deterministic=true)
365 get_sample_set(input_sample_channel, completed_sample_channel,
366     → progress_channel)
367
368
369 #collect sample dfs by partition id when ready
370 collected_counter = 0

```

```

339     sample_record_dfs = Dict{String,DataFrame}()
340     while collected_counter < partitions
341         wait(completed_sample_channel)
342         partition_id, sample_df = take!(completed_sample_channel)
343         sample_record_dfs[partition_id] = sample_df
344         collected_counter += 1
345     end
346
347     #get_sample entire feature selection
348     coordinate_partitions = partition_genome_coordinates(gff,
349     ↳ perigenic_pad)
350     partition_id="exon"
351     partition=coordinate_partitions[partition_id]
352     add_metacoordinates!(partition)
353     metacoordinate_bitarray=true(partition.MetaEnd[end])
354     scaffold_sequence_record_dict = build_scaffold_seq_dict(genome,
355     ↳ index)
356
357     #test fetch_sequence strand char ArgumentError
358     @test_throws ArgumentError fetch_sequence("1",
359     ↳ Dict{String,LongSequence}("1"⇒LongSequence{DNAAlphabet{2}}("AAAAAAAAAAAAAA"))
360     ↳ 1, 5, 'L')
361
362     #test get_feature_row_index ArgumentError
363     @test_throws DomainError get_feature_row_index(partition,0)
364
365     @test length(get_sample(metacoordinate_bitarray, 1, 250, partition,
366     ↳ scaffold_sequence_record_dict)[6])=60
367
368     synthetic_seq =
369     ↳ BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())
370     for (partid, df) in sample_record_dfs
371         for sample in eachrow(df)
372             target_seq =
373             ↳ synthetic_seq[sample.SampleStart:sample.SampleEnd]
374             strand = sample.Strand
375             if sample.Strand == '-'
376                 target_seq = reverse_complement(target_seq)
377             end
378             @test sample.SampleSequence == target_seq
379         end
380     end
381
382 
```

```

375 #execute_sample_jobs API
376 wkpool=addprocs(1)
377 @everywhere using BioBackgroundModels
378
379 channels = setup_sample_jobs(genome, index, gff, sample_set_length,
380   ↳ min_sample_window, max_sample_window, perigenic_pad;
381   ↳ deterministic=true)
380 records=execute_sample_jobs(channels, wkpool)
381
382 rmprocs(wkpool)
383 end
384
385 @testset "likelihood_funcs/hmm.jl & bg_lh_matrix.jl functions" begin
386     pvec = [.4,.3,.2,.1]
387     trans = ones(1,1)
388     B = [Categorical(pvec)]
389     hmm = BHMM(trans, B)
390
391     testseq=zeros(Int64,1,5)
392     testseq[1:4] = [1,2,3,4]
393     @test isapprox(get_BGHMM_symbol_lh(testseq, hmm)[1], log.(pvec[1]))
394     @test
395         ↳ isapprox(obs_lh_given_hmm(testseq,hmm),obs_lh_given_hmm(testseq,hmm,linear=false))
396     @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
397         ↳ obs_lh_given_hmm(testseq,hmm))
398     @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
399         ↳ obs_lh_given_hmm(testseq,hmm,linear=false))
400
401     pvec=[.25,.25,.25,.25]
402     trans = [.9 .1
403               .1 .9]
404     B = [Categorical(pvec), Categorical(pvec)]
405     hmm = BHMM(trans, B)
406
407     @test
408         ↳ isapprox(obs_lh_given_hmm(testseq,hmm),obs_lh_given_hmm(testseq,hmm,linear=false))
409     @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
410         ↳ obs_lh_given_hmm(testseq,hmm))
411     @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
412         ↳ obs_lh_given_hmm(testseq,hmm,linear=false))
413
414     testseq=zeros(Int64,1,1001)
415     testseq[1,1:1000]=rand(1:4,1000)

```

```

410
411     @test isapprox(sum(get_BGHMM_symbol_lh(testseq,
412         → hmm)),obs_lh_given_hmm(testseq,hmm))
413
414     Dex = [Categorical([.3, .1, .3, .3]),Categorical([.15, .35, .35,
415         → .15])]
416     Dper = [Categorical([.15, .35, .35, .15]),Categorical([.4, .1, .1,
417         → .4])]
418     Dint = [Categorical([.4, .1, .1, .4]),Categorical([.45, .05, .05,
419         → .45])]
420     BGHMM_dict = Dict{String,BHMM}()
421     BGHMM_dict["exon"] = BHMM(trans, Dex)
422     BGHMM_dict["perixonic"] = BHMM(trans, Dper)
423     BGHMM_dict["intergenic"] = BHMM(trans, Dint)
424
425     position_length=141;perigenic_pad=250;
426     position_df = make_padded_df(posfasta, gff, genome, index,
427         → position_length)
428
429     reader=open(FASTA.Reader, genome, index=index)
430     seq=FASTA.sequence(reader["CM002885.2.1"])
431     @test position_df.Start[1]==501
432     @test position_df.End[1]==641
433     @test position_df.PadSeq[1]==seq[360:641]
434     @test position_df.PadStart[1]==360
435     @test position_df.RelStart[1]==141
436     @test position_df.SeqOffset[1]==0
437
438     offset_position_df = make_padded_df(posfasta, gff, genome, index,600)
439     @test offset_position_df.Start[1]==501
440     @test offset_position_df.End[1]==641
441     @test offset_position_df.PadSeq[1]==seq[1:641]
442     @test offset_position_df.PadStart[1]==1
443     @test offset_position_df.RelStart[1]==500
444     @test offset_position_df.SeqOffset[1]==100
445
446     add_partition_masks!(position_df, gff, perigenic_pad)
447     @test all(position_df.MaskMatrix[1][1:151,1]==2)
448     @test all(position_df.MaskMatrix[1][152:211,1]==3)
449     @test all(position_df.MaskMatrix[1][212:241,1]==2)
450     @test all(position_df.MaskMatrix[1][242:end,1]==3)
451     @test all(position_df.MaskMatrix[1][1:151,2]==-1)
452     @test all(position_df.MaskMatrix[1][152:211,2]==1)

```

```

448 @test all(position_df.MaskMatrix[1][212:end,2].== -1)
449
450 lh_matrix=BGHMM_likelihood_calc(position_df,BGHMM_dict)
451 @test size(lh_matrix)=(position_length*2,1)
452
453
454     → periexonic_frag=reverse_complement!(LongSequence{DNAAlphabet{2}})(seq[360:510])
455 pno=get_order_n_seqs([periexonic_frag],0)
456 pcode=code_seqs(pno)
457 plh=get_BGHMM_symbol_lh(pcode, BGHMM_dict["periexonic"])
458 @test isapprox(lh_matrix[1:151,1], reverse(plh))
459
460 exonic_frag=LongSequence{DNAAlphabet{2}}(seq[511:570])
461 eno=get_order_n_seqs([exonic_frag],0)
462 ecode=code_seqs(eno)
463 elh=get_BGHMM_symbol_lh(ecode, BGHMM_dict["exon"])
464 @test isapprox(lh_matrix[152:211,1],elh)
465 end
466
467 @testset "EM/baum-welch.jl MS_HMMBase equivalency and functions" begin
468     A = [.5 .5
469             .5 .5]
470     B = [Categorical(ones(4)/4), Categorical([.7,.1,.1,.1])]
471     hmm = BHMM(A, B)
472     log_A = log.(hmm.A)
473
474     obs = zeros(UInt8, 22,1)
475     obs[1:21] = [4,3,3,2,3,2,1,1,2,3,3,3,4,4,2,3,2,3,4,3,2]
476     obs_lengths=[21]
477
478     lls = BioBackgroundModels.bw_llhs(hmm,obs)
479     m_lls = mouchet_log_likelihoods(hmm,obs[1:21])
480
481     K,Tmaxplus1,0 = size(lls)
482     T=Tmaxplus1-1
483
484     #TEST LOG LIKELIHOOD FN
485     @test transpose(lls[:,1:end-1,1]) == m_lls
486
487     log_α = BioBackgroundModels.messages_forwards_log(hmm.a, hmm.A, lls,
488             → obs_lengths)
489     m_log_α = mouchet_messages_forwards_log(hmm.a, hmm.A, m_lls)
490
491

```

```

489 #TEST FORWARD MESSAGES FN
490 @test transpose(log_α[:,1:end-1,1]) = m_log_α
491
492 log_β = BioBackgroundModels.messages_backwards_log(hmm.A, lls,
493   ↪ obs_lengths)
493 m_log_β = mouchet_messages_backwards_log(hmm.A, m_lls)
494
495 #TEST BACKWARDS MESSAGES FN
496 @test isapprox(transpose(log_β[:,1:end-1,1]), m_log_β)
497
498 lls = permutedims(lls, [2,3,1]) # from (K,T,O) to (T,O,K)
499 log_α = permutedims(log_α, [2,3,1])
500 log_β = permutedims(log_β, [2,3,1])
501
502 # "Testing obs probability calcs ... "
503 #TEST OBSERVATION PROBABILITY CALC
504 normalizer = logsumexp(m_log_α[1,:]+m_log_β[1,:])
505 log_pobs = logsumexp(lps.(log_α[1,1,:], log_β[1,1,:]))
506
507 @test isapprox(normalizer, log_pobs)
508
509 # "Testing ξ, γ calcs ... "
510
511 log_ξ = fill(-Inf, Tmaxplus1, 0, K, K)
512 log_γ = fill(-Inf, Tmaxplus1, 0, K)
513
514 m_log_ξ = zeros(T, K, K)
515
516 for t = 1:T-1, i = 1:K, j = 1:K
517   m_log_ξ[t,i,j] = m_log_α[t,i] + log_A[i,j] + m_log_β[t+1,j] +
518     ↪ m_lls[t+1,j] - normalizer
519 end
520
521 for i = 1:K, j = 1:K, o = 1:O
522   obsl = obs_lengths[o]
523   for t = 1:obsl-1 #log_ξ & log_γ calculated to T-1 for each o
524     log_ξ[t,o,i,j] = lps(log_α[t,o,i], log_A[i,j], log_β[t+1,o,j],
525       ↪ lls[t+1,o,j], -log_pobs[o])
526     log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
527   end
528   t=obsl #log_ξ @ T = 0
529   log_ξ[t,o,i,j] = 0
530   log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs)

```

```

529     end
530
531     ξ = exp.(log_ξ)
532     k_ξ = sum(ξ, dims=[3,4])
533     nan_mask = k_ξ .== 0; k_ξ[nan_mask] .= Inf #prevent NaNs in dummy
      ↵ renorm arising from multiple sequences indexing
534     ξ ./= k_ξ #dummy renorm across K to keep numerical creep from
      ↵ causing isprobvec to fail on new new_A during hmm creation
535
536     m_ξ = exp.(m_log_ξ)
537     m_ξ ./= sum(m_ξ, dims=[2,3])
538
539     #check equivalency of xi calc methods
540     @test isapprox(reshape(ξ,Tmaxplus1,K,K)[1:21,:,:],m_ξ)
541
542     γ = exp.(log_γ)
543     k_γ = sum(γ, dims=3); k_γ[nan_mask[:, :, :]] .= Inf #prevent NaNs in
      ↵ dummy renorm
544     γ ./= k_γ
545
546     m_γ = exp.((m_log_α .+ m_log_β) .- normalizer)
547
548     #check equivalency of gamma calculations
549     @test isapprox(reshape(γ,Tmaxplus1,K)[1:21,:,:],m_γ)
550
551     # "Testing initial and transition matrix calcs ... "
552
553     m_new_A = sum(m_ξ[1:end-1,:,:], dims=1)[1,:,:]
554     m_new_A ./= sum(m_new_A, dims=2)
555
556     new_A = zeros(K,K)
557     for i=1:K, j=1:K
558         Σotξ_vec = zeros(0)
559         Σotγ_vec = zeros(0)
560         for o in 1:0
561             Σotξ_vec[o] = sum(ξ[1:obs_lengths[o]-1,o,i,j])
562             Σotγ_vec[o] = sum(γ[1:obs_lengths[o]-1,o,i])
563         end
564         new_A[i,j] = sum(Σotξ_vec) / sum(Σotγ_vec)
565     end
566     new_A ./= sum(new_A, dims=[2]) #dummy renorm
567
568     #check equivalency of transition matrix calculations

```



```

609  #verify that above methods independently produce equivalent output,
  ↵  and that this is true of multiple identical obs, but not true of
  ↵  different obs sets
610  mouchet_hmm = mouchet_mle_step(hmm, obs[1:21])
611
612  new_hmm = BioBackgroundModels.bw_step(hmm, obs, obs_lengths)
613
614  dblobs = zeros(UInt8, 22,2)
615  dblobs[1:21,1] = [4,3,3,2,3,2,1,1,2,3,3,3,4,4,2,3,2,3,4,3,2]
616  dblobs[1:21,2] = [4,3,3,2,3,2,1,1,2,3,3,3,4,4,2,3,2,3,4,3,2]
617  dblobs_lengths=[21,21]
618  dbl_hmm = BioBackgroundModels.bw_step(hmm, dblobs, dblobs_lengths)
619
620  otherobs = dblobs
621  otherobs[1:21,2] = [1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1]
622  other_hmm = BioBackgroundModels.bw_step(hmm, otherobs,
  ↵  dblobs_lengths)
623
624  for n in fieldnames(typeof(new_hmm[1]))
625      if n == :B
626          for (d, dist) in enumerate(new_hmm[1].B)
627              @test dist.support == mouchet_hmm[1].B[d].support
628              @test isapprox(dist.p,mouchet_hmm[1].B[d].p)
629              @test dist.support == dbl_hmm[1].B[d].support
630              @test isapprox(dist.p,dbl_hmm[1].B[d].p)
631              @test dist.support == other_hmm[1].B[d].support
632              @test !isapprox(dist.p, other_hmm[1].B[d].p)
633          end
634      elseif n == :partition
635          @test getfield(new_hmm[1],n) == getfield(mouchet_hmm[1],n)
636          @test getfield(new_hmm[1],n) == getfield(dbl_hmm[1],n)
637          @test getfield(new_hmm[1],n) == getfield(other_hmm[1],n)
638      else
639          @test isapprox(getfield(new_hmm[1],n),
  ↵  getfield(mouchet_hmm[1],n))
640          @test isapprox(getfield(new_hmm[1],n),
  ↵  getfield(dbl_hmm[1],n))
641          @test !isapprox(getfield(new_hmm[1],n),
  ↵  getfield(other_hmm[1],n))
642      end
643  end
644
645  @test new_hmm[2] == mouchet_hmm[2] ≠ dbl_hmm[2] ≠ other_hmm[2]

```

```

646
647 # "Testing fit_mle! ... "
648
649 # "test fit_mle! function"
650 input_hmms= RemoteChannel(()→Channel{Tuple}(1))
651 output_hmms = RemoteChannel(()→Channel{Tuple}(30))
652 chainid=Chain_ID("Test",2,0,1)
653 put!(input_hmms, (chainid, 2, hmm, 0.0, transpose(obs)))
654 BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
655     → EM_func=BioBackgroundModels.bw_step, max_iterates=4,
656     → verbose=true)
657 wait(output_hmms)
658 workerid, jobid, iterate, hmm3, log_p, epsilon, converged =
659     → take!(output_hmms)
660 @test jobid == chainid
661 @test iterate == 3
662 @test assert_hmm(hmm3.a, hmm3.A, hmm3.B)
663 @test size(hmm3) == size(hmm) == (2,4)
664 @test log_p < 1
665 @test log_p == obs_lh_given_hmm(transpose(obs),hmm, linear=false)
666 @test converged == false
667 wait(output_hmms)
668 workerid, jobid, iterate, hmm4, log_p, epsilon, converged =
669     → take!(output_hmms)
670 @test jobid == chainid
671 @test iterate == 4
672 @test assert_hmm(hmm4.a, hmm4.A, hmm4.B)
673 @test size(hmm4) == size(hmm) == (2,4)
674 @test log_p < 1
675 @test log_p == obs_lh_given_hmm(transpose(obs),hmm3,linear=false)
676 @test converged == false
677
678 # "Test convergence.. "
679 obs=zeros(UInt8, 101, 2)
680 for i in 1:size(obs)[2]
681     obs[1:100,i]=rand(1:4,100)
682 end
683 input_hmms= RemoteChannel(()→Channel{Tuple}(1))
684 output_hmms = RemoteChannel(()→Channel{Tuple}(30))
685 put!(input_hmms, (chainid, 2, hmm, 0.0, transpose(obs)))
686 BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
687     → EM_func=BioBackgroundModels.bw_step, delta_thresh=.05,
688     → max_iterates=100, verbose=true)

```

```

683     wait(output_hmms)
684     workerid, jobid, iterate, hmm4, log_p, epsilon, converged =
685         → take!(output_hmms)
686     while isready(output_hmms)
687         workerid, jobid, iterate, hmm4, log_p, epsilon, converged =
688             → take!(output_hmms)
689     end
690     @test converged==1
691
692     #test t_add_categorical_counts DimensionMismatch
693     @test_throws DimensionMismatch t_add_categorical_counts!(zeros(4),
694         → zeros(UInt8,2,3), zeros(2,2))
695
696 end
697
698 @testset "EM/churbanov.jl Baum-Welch equivalency and functions" begin
699     # "Setting up for MLE function tests.."
700     A = fill((1/6),6,6)
701     B = [Categorical(ones(4)/4),
702         → Categorical([.7,.05,.15,.1]),Categorical([.15,.35,.4,.1]),
703         → Categorical([.6,.15,.15,.1]),Categorical([.1,.4,.4,.1]),
704         → Categorical([.2,.2,.3,.3])]
705     hmm = BHMM(A, B)
706     log_A = log.(hmm.A)
707
708     obs = zeros(Int64,1,250)
709     obs[1:249] = rand(1:4,249)
710     obs_lengths=[249]
711     # "Testing mle_step ... "
712
713     #verify that above methods independently produce equivalent output,
714     → and that this is true of multiple identical obs, but not true of
715     → different obs sets
716     mouchet_hmm = mouchet_mle_step(hmm, obs[1:249])
717
718     new_hmm = linear_step(hmm, obs, obs_lengths)
719
720     ms_sng = BioBackgroundModels.bw_step(hmm, Array(transpose(obs)),
721         → obs_lengths)
722
723     dblobs = zeros(Int64, 2,250)
724     dblobs[1,1:249] = obs[1:249]
725     dblobs[2,1:249] = obs[1:249]

```

```

717     dbllobs_lengths=[249,249]
718     dbl_hmm = linear_step(hmm, dbllobs, dbllobs_lengths)
719
720
721     ms_dbl =BioBackgroundModels.bw_step(hmm,
722         → Array(transpose(dbllobs)),dblobs_lengths)
723
724     otherobs = deepcopy(dbllobs)
725     otherobs[2,1:249] = rand(1:4,249)
726     if otherobs[1,1]==otherobs[2,1]
727         otherobs[1,1]=1&&(otherobs[2,1]==2)
728         otherobs[1,1]=2&&(otherobs[2,1]==3)
729         otherobs[1,1]=3&&(otherobs[2,1]==4)
730         otherobs[1,1]=4&&(otherobs[2,1]==1)
731     end
732
733     other_hmm = linear_step(hmm, otherobs, dbllobs_lengths)
734
735     ms_hmm = BioBackgroundModels.bw_step(hmm,
736         → Array(transpose(otherobs)),dblobs_lengths)
737
738     for n in fieldnames(typeof(new_hmm[1]))
739         if n == :B
740             for (d, dist) in enumerate(new_hmm[1].B)
741                 @test dist.support==mouchet_hmm[1].B[d].support
742                 @test isapprox(dist.p,mouchet_hmm[1].B[d].p)
743                 @test dist.support==ms_sng[1].B[d].support
744                 @test isapprox(dist.p,ms_sng[1].B[d].p)
745                 @test dist.support==dbl_hmm[1].B[d].support
746                 @test isapprox(dist.p,dbl_hmm[1].B[d].p)
747                 @test dist.support==other_hmm[1].B[d].support
748                 @test ms_hmm[1].B[d].support==other_hmm[1].B[d].support
749                 @test !isapprox(dist.p, other_hmm[1].B[d].p)
750                 @test isapprox(ms_hmm[1].B[d].p, other_hmm[1].B[d].p)
751             end
752         elseif n == :partition
753             @test getfield(new_hmm[1],n) == getfield(mouchet_hmm[1],n)
754             @test getfield(new_hmm[1],n) == getfield(dbl_hmm[1],n)
755             @test getfield(new_hmm[1],n) == getfield(other_hmm[1],n)
756         else
757

```

```

758         @test isapprox(getfield(new_hmm[1],n),
759                         getfield(mouchet_hmm[1],n))
760
761         @test isapprox(getfield(new_hmm[1],n),
762                         getfield(dbl_hmm[1],n))
763
764         @test !isapprox(getfield(new_hmm[1],n),
765                         getfield(other_hmm[1],n))
766
767     end
768
769
770     @test ms_sng[2] == new_hmm[2] == mouchet_hmm[2] != dbl_hmm[2] ==
771         → ms_dbl[2] != other_hmm[2] == ms_hmm[2]
772
773     # "Testing fit_mle! ... "
774
775     #test fit_mle! function
776     input_hmms= RemoteChannel(()→Channel{Tuple}(1))
777     output_hmms = RemoteChannel(()→Channel{Tuple}(30))
778     chainid=Chain_ID("Test",6,0,1)
779     put!(input_hmms, (chainid, 2, hmm, 0.0, obs))
780     BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
781         → max_iterates=4, verbose=true)
782     wait(output_hmms)
783     workerid, jobid, iterate, hmm3, log_p, delta, converged =
784         → take!(output_hmms)
785     @test jobid == chainid
786     @test iterate == 3
787     @test BioBackgroundModels.assert_hmm(hmm3.a, hmm3.A, hmm3.B)
788     @test size(hmm3) == size(hmm) == (6,4)
789     @test log_p < 1
790     @test log_p == linear_likelihood(obs, hmm)
791     @test converged == false
792     wait(output_hmms)
793     workerid, jobid, iterate, hmm4, log_p, delta, converged =
794         → take!(output_hmms)
795     @test jobid == chainid
796     @test iterate == 4

```

```

793     @test BioBackgroundModels.assert_hmm(hmm4.a, hmm4.A, hmm4.B)
794     @test size(hmm4) == size(hmm) == (6,4)
795     @test log_p < 1
796     @test log_p == linear_likelihood(obs, hmm3)
797     @test converged == false

798
799     # "Test convergence.."
800     obs=zeros(Int64, 4, 1001)
801     for o in 1:size(obs)[1]
802         obsl=rand(100:1000)
803         obs[o,1:obsl]=rand(1:4,obsl)
804     end
805     input_hmms= RemoteChannel(()→Channel{Tuple}(1))
806     output_hmms = RemoteChannel(()→Channel{Tuple}(Inf))
807     put!(input_hmms, (chainid, 2, hmm, 0.0, obs))
808     BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
809         → delta_thresh=.05, max_iterates=100, verbose=true)
810     wait(output_hmms)
811     workerid, jobid, iterate, hmm4, log_p, delta, converged =
812         → take!(output_hmms)
813     while isready(output_hmms)
814         workerid, jobid, iterate, hmm4, log_p, delta, converged =
815             → take!(output_hmms)
816     end
817     @test converged==1
818
819     @testset "API/EM_master.jl and reports.jl API tests" begin
820         #CONSTANTS FOR BHMM LEARNING
821         replicates = 4 #repeat optimisation from this many seperately
822             → initialised samples from the prior
823         Ks = [1,2,4,6] #mosaic class #s to test
824         order_nos = [0,1,2] #DNA kmer order #s to test
825         sample_set_length=100
826         min_sample_window=5
827         max_sample_window=25
828         perigenic_pad=250
829
830         wkpool=addprocs(2)
831         @everywhere using BioBackgroundModels

```

```

830     channels = setup_sample_jobs(genome, index, gff, sample_set_length,
831         ↳ min_sample_window, max_sample_window, perigenic_pad;
832         ↳ deterministic=true)
833     sample_record_dfs=execute_sample_jobs(channels, wkpool)
834     training_sets, test_sets = split_obs_sets(sample_record_dfs)
835
836
837     job_ids=Vector{Chain_ID}()
838     for (obs_id, obs) in training_sets, K in Ks, order in order_nos, rep
839         ↳ in 1:replicates
840         push!(job_ids, Chain_ID(obs_id, K, order, rep))
841     end
842
843     no_input_hmms, chains, input_hmms, output_hmms =
844         ↳ setup_EM_jobs!(job_ids, training_sets)
845     @test no_input_hmms ==
846         ↳ replicates*length(Ks)*length(order_nos)*length(training_sets)
847
848     while isready(input_hmms)
849         jobid, start_iterate, hmm, last_norm, observations =
850             ↳ take!(input_hmms)
851         @test last_norm == linear_likelihood(observations, hmm)
852         obs_lengths = [findfirst(iszero, observations[o,:])-1 for o in
853             ↳ 1:size(observations)[1]]
854         #make sure input HMMs are valid and try to mle_step them and
855             ↳ ensure their 1-step children are valid
856         @test assert_hmm(hmm.a, hmm.A, hmm.B)
857         new_hmm, prob = linear_step(hmm, observations, obs_lengths)
858         @test assert_hmm(hmm.a, hmm.A, hmm.B)
859         @test prob < 0
860     end
861
862     genome_reader = open(FASTA.Reader, genome, index=index)
863
864         ↳ seq=LongSequence{DNAAlphabet{2}}(FASTA.sequence(genome_reader["CM002885.2.1"]))
865     seqdict = Dict("test"⇒[seq for i in 1:3])
866     seqdict["blacklist"] = seqdict["test"]
867
868     job_ids=Vector{Chain_ID}()
869     push!(job_ids, Chain_ID("blacklist", 1, 0, 1))
870     push!(job_ids, Chain_ID("blacklist", 1, 0, 2))

```

```

864 Ks=[1,2]; order_nos=[0,1]
865 for K in Ks, order in order_nos, replicate in 1:2
866     push!(job_ids, Chain_ID("test", K, order, replicate))
867 end
868
869 wkpool=addprocs(2, topology=:master_worker)
870 @everywhere using BioBackgroundModels
871
872 load_dict=Dict{Int64,LoadConfig}()
873 for (n,wk) in enumerate(wkpool)
874     n==1 &&
875         (load_dict[wk]=LoadConfig(1:2,0:1,blacklist=[Chain_ID("blacklist",1,0,1)])
876     n==2 && (load_dict[wk]=LoadConfig(1:2,0:1))
877 end
878 #test work resumption, load configs
879
880 em_jobset=setup_EM_jobs!(job_ids, seqdict, delta_thresh=10000.)
881 execute_EM_jobs!(wkpool, em_jobset..., "testchains",
882     → delta_thresh=10000., verbose=true, load_dict=load_dict)
883
884 for (chain_id,chain) in em_jobset[2]
885     @test chain[end].converged = true
886     @test chain[end].delta ≤ 10000
887     @test typeof(chain[end].hmm)=BHMM{Float64}
888     @test 1 ≤ chain[end].iterate ≤ 5000
889 end
890
891 wkpool=addprocs(2, topology=:master_worker)
892 @everywhere using BioBackgroundModels
893
894 em_jobset=setup_EM_jobs!(job_ids, seqdict,
895     → chains=deserialize("testchains"), delta_thresh=.1)
896 execute_EM_jobs!(wkpool, em_jobset..., "testchains", delta_thresh=.1,
897     → verbose=true)
898
899 for (chain_id,chain) in em_jobset[2]
900     @test chain[end].converged = true
901     @test chain[end].delta ≤ .1
902     @test typeof(chain[end].hmm)=BHMM{Float64}
903     @test 1 ≤ chain[end].iterate ≤ 5000
904 end
905
906 #test for already converged warning

```

```

903     wkpool=addprocs(2, topology=:master_worker)
904     @everywhere using BioBackgroundModels
905     @test_throws ArgumentError execute_EM_jobs!(wkpool, em_jobset..., 
906         → "testchains", delta_thresh=.01, verbose=true)
907
908     rm("testchains")
909
910     @test_throws ArgumentError
911         → generate_reports(Dict{Chain_ID,Vector{EM_step}}(),seqdict)
912     @test_throws ArgumentError
913         → generate_reports(em_jobset[2],Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}
914
915     @test_throws ArgumentError
916         → BioBackgroundModels.report_chains(em_jobset[2],Dict{String,Vector{LongSequence{DNAAlphabet{2}}}})
917
918     report_folders=generate_reports(em_jobset[2],seqdict)
919     folder=report_folders["test"]
920     @test "test"==folder.partition_id
921     show(folder.partition_report)
922     show(folder.replicate_report)
923     for id in folder.partition_report.best_repset
924         show(folder.chain_reports[id])
925     end
926
927     end
928
929     rm(genome)
930     rm(index)
931     rm(gff)
932     rm(posfasta)

```

15.2.28 /test/synthetic_sequence_gen.jl

```

1 function print_synthetic_fasta(path::String,line_length::Integer=80)
2     header=>CM002885.2.1 BioBackgroundModels Synthetic chromosome
3         → for tests\n"
4     write(path, header,
5         → format_lines(generate_synthetic_seq(),line_length))
6 end
7
8 function print_synthetic_index(path::String)
9     write(path,
10        → "CM002885.2.1          1000          5          80          81\n")

```



```

42 function generate_synthetic_seq(gene_start::Integer=501,
43   ↪ UTR3L::Integer=10, exon_length::Integer=60,
44   ↪ intron_length::Integer=40, no_exons::Integer=5, UTR5L::Integer=25,
45   ↪ OAL::Integer=1000,
46   ↪ (iseq,pseq,eseq)::Tuple{String,String,String}=( "AT", "CG", "CAT" );
47   ↪ verbose::Bool=false)
48
49     @assert length(iseq) == 2
50     @assert length(pseq) == 2
51     @assert length(eseq) == 3
52     @assert mod(gene_start-1,2) == 0
53     @assert mod(exon_length,3) == 0
54     @assert mod(intron_length,2) == 0
55     gene_length=gene_start+UTR3L+(no_exons*exon_length)+((no_exons-1)*intron_length)
56     @assert gene_length ≤ OAL
57     seq=""
58     for i in 1:((gene_start-1) / 2)
59       seq *= iseq
60     end
61     verbose && @info "Intergenic length $(length(seq))"
62     for i in 1:UTR3L / 2
63       seq *= pseq
64     end
65     verbose && @info "Added 3'UTR ... $(length(seq))"
66     for ex in 1:no_exons-1
67       for i in 1:(exon_length/3)
68         seq*=eseq
69       end
70       ex == 1 ? ilength = 30 : ilength = 40
71       for i in 1:(ilength/2)
72         seq*=pseq
73       end
74     end
75     for i in 1:(exon_length/3)
76       seq*=eseq
77     end
78     verbose && @info "Added exons and introns ... $(length(seq))"
79     for i in 1:UTR5L / 2
80       seq *= pseq
81     end
82     verbose && @info "Added 5'UTR ... $(length(seq))"
83
84     for i in 1:((OAL-length(seq))/2)
85       seq*=iseq
86   end

```

```

80     end
81
82     verbose && @info "Generated synthetic genome sequence of length
83         → $(length(seq))"
84
85     return seq
86 end
87
88 function format_lines(seq::String, line_length::Integer)
89     a=join((SubString(seq,i,min(i+line_length-1,length(seq))) for
90             → i=1:line_length:length(seq)), '\n')
91     return(a)
92 end

```

15.3 BioMotifInference

15.3.1 /README.md

```

1 ## BioMotifInference
2 [![Build
3     → Status](https://travis-ci.org/mmattocks/BioMotifInference.jl.svg?branch=master)](https://travis-ci.org/mmattocks/BioMotifInference.jl)
4 [![codecov](https://codecov.io/gh/mmattocks/BioMotifInference.jl/branch/master/graph/badge.svg)](https://codecov.io/gh/mmattocks/BioMotifInference.jl)
5 [![Project Status: WIP – Initial development is in progress, but there
6     → has not yet been a stable, usable release suitable for the
7     → public.](https://www.repostatus.org/badges/latest/wip.svg)](https://www.repostatus.org/badges/latest/wip.svg)

```

15.3.2 /src/BioMotifInference.jl

```

1 module BioMotifInference
2     using BioBackgroundModels, BioSequences, Distributed, Distributions,
3         → Serialization, UnicodePlots
4     import DataFrames:DataFrame
5     import ProgressMeter: AbstractProgress, Progress, @showprogress,
6         → next!, move_cursor_up_while_clearing_lines, printover,
7         → durationstring
8     import Printf: @sprintf
9     import StatsFuns: logaddexp, logsumexp #both are needed as logsumexp
10        → for two terms is deprecated
11     import Random: rand, seed!, shuffle!
12     import Distances: euclidean

```

```

9
10 #CONSTANTS AND PERMUTE FUNCTION ARGUMENT DEFAULTS GIVING RISE TO
11   ↳ IMPLEMENTATION-SPECIFIC SAMPLING EFFECTS
12 global MOTIF_EXPECT=1. #motif expectation- this value to be divided
13   ↳ by obs lengths to obtain penalty factor for scoring
14
15 global REVCOMP=true #calculate scores on both strands?
16
17 global TUNING_MEMORY=20 #coefficient multiplied by Permute_Instruct
18   ↳ function call limit to give total number of calls remembered by
19   ↳ tuner per function
20 global CONVERGENCE_MEMORY=500 #number of iterates to display for
21   ↳ convergence interval history
22
23 global SRC_PERM_FREQ=.5 #frequency with which random_decorrelate will
24   ↳ permute a source
25
26 global PWM_SHIFT_DIST=Weibull(.5,.1) #distribution of weight matrix
27   ↳ permutation magnitudes
28 global PWM_SHIFT_FREQ=.2 #proportion of positions in source to
29   ↳ permute weight matrix
30 global PWM_LENGTHPERM_FREQ=.2 #proportion of sources to permute
31   ↳ length
32 global LENGTHPERM_RANGE=1:3
33
34 global PRIOR_WT=3. #estimate prior dirichlets from product of this
35   ↳ constant and sample "mle" wm
36 global PRIOR_LENGTH_MASS=.8
37
38 global EROSION_INFO_THRESH=1.
39
40 global CONSOLIDATE_THRESH=.035
41
42 include("IPM/ICA_PWM_Model.jl")
43 export Model_Record
44 export ICA_PWM_Model
45 include("IPM/IPM_likelihood.jl")
46 include("IPM/IPM_prior_utilities.jl")
47 assemble_source_priors
48 include("ensemble/IPM_Engine.jl")
49 export IPM_Engine, assemble_IPMs
50 include("permutation/permute_utilities.jl")
51 include("permutation/orthogonality_helper.jl")

```

```

42   include("permutation/permute_functions.jl")
43   export full_perm_funcvec
44   include("permutation/permute_control.jl")
45   export Permute_Instruct
46   include("permutation/Permute_Tuner.jl")
47   include("ensemble/ensemble_utilities.jl")
48   export ensemble_history, reset_ensemble!, move_ensemble!,
49     ↳ copy_ensemble!, rewind_ensemble, complete_evidence, get_model,
50     ↳ show_models
51   include("utilities/model_display.jl")
52   include("utilities/worker_diagnostics.jl")
53   include("utilities/ns_progressmeter.jl")
54   include("utilities/synthetic_genome.jl")
55   include("utilities/worker_sequencer.jl")
56   export synthetic_sample
57   include("nested_sampler/nested_step.jl")
58   include("nested_sampler/converge_ensemble.jl")
59   export converge_ensemble!
58
59 end # module

```

15.3.3 /src/IPM/ICA_PWM_Model.jl

```

1 struct Model_Record #record struct to associate a log_Li with a saved,
2   ↳ calculated model
3   path::String
4   log_Li::AbstractFloat
5 end
6
6 struct ICA_PWM_Model #Independent component analysis position weight
7   ↳ matrix model
8   name::String #designator for saving model to posterior
9   origin::String #functions instantiating IPMs should give an
10    ↳ informative desc of the means by which the model was generated
11   sources::Vector{Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}}
12    ↳ #vector of PWM signal sources (LOG PROBABILITY!!!) tupled with an
13    ↳ index denoting the position of the first PWM base on the prior
14    ↳ matrix- allows us to permute length and redraw from the
15    ↳ appropriate prior position
16   source_length_limits::UnitRange{<:Integer} #min/max source lengths
17    ↳ for init and permutation
18   mix_matrix::BitMatrix # obs x sources bool matrix

```

```

12   log_Li::AbstractFloat
13   permute_blacklist::Vector{Function} #blacklist of functions that
14     → ought not be used to permute this model (eg. because to do so
15     → would not generate a different model for IPMs produced from
16     → fitting the mix matrix)
17   function ICA_PWM_Model(name, origin, sources, source_length_limits,
18     → mix_matrix, log_Li, permute_blacklist=Vector{Function}())
19     verify_srcs(name, origin, sources)
20     new(name, origin, sources, source_length_limits, mix_matrix,
21       → log_Li, permute_blacklist)
22   end
23 end
24
25 #ICA_PWM_Model FUNCTIONS
26 ICA_PWM_Model(name :: String,
27   → source_priors :: AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat},<:AbstractVector{<:Union{<:BitMatrix,<:AbstractFloat},<:AbstractFloat},<:AbstractArray{<:AbstractFloat},<:AbstractArray{<:Integer}},<:UnitRange{<:Integer}}}} = init_IPM(name,
28   → mix_prior :: Tuple{BitMatrix,<:AbstractFloat},
29   → bg_scores :: AbstractArray{<:AbstractFloat},
30   → observations :: AbstractArray{<:Integer},
31   → source_length_limits :: UnitRange{<:Integer}) = init_IPM(name,
32   → source_priors, mix_prior, bg_scores, observations, source_length_limits)
33
34 function verify_srcs(name, ori,
35   → sources :: Vector{<:Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}})
36   for (n,(pwm,pi)) in enumerate(sources)
37     !all(isprobvec(exp.(pwm[pos,:])) for pos in 1:size(pwm,1)) &&
38     → throw(DomainError("Bad probvec in model $name, source $n,
39     → origin $(ori):$(exp.(pwm))"))
40   end
41 end
42
43 #MODEL INIT
44 function init_IPM(name :: String,
45   → source_priors :: AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat},<:AbstractVector{<:Union{<:BitMatrix,<:AbstractFloat},<:AbstractFloat},<:AbstractArray{<:AbstractFloat},<:AbstractArray{<:Integer}},<:UnitRange{<:Integer}}}}),
46   → mix_prior :: Tuple{BitMatrix,<:AbstractFloat},
47   → bg_scores :: AbstractArray{<:AbstractFloat},
48   → observations :: AbstractArray{<:Integer},
49   → source_length_limits :: UnitRange{<:Integer})
50   assert_obs_bg_compatibility(observations, bg_scores)
51   T,O = size(observations)
52   S=length(source_priors)
53   obs_lengths=[findfirst(iszero, observations[:,o])-1 for o in
54     → 1:size(observations)[2]]

```

```

35 sources=init_logPWM_sources(source_priors, source_length_limits)
36 mix=init_mix_matrix(mix_prior,0,S)
37 log_lh = IPM_likelihood(sources, observations, obs_lengths,
38   ↳ bg_scores, mix)
39
40 return ICA_PWM_Model(name, "init_IPM", sources, source_length_limits,
41   ↳ mix, log_lh)
42 end

43 #init_IPM SUBFUNCS
44 function assert_obs_bg_compatibility(obs, bg_scores)
45   T,O=size(obs)
46   t,o=size(bg_scores)
47   O!=o && throw(DomainError("Background scores and
48     ↳ observations must have same number of observation
49     ↳ columns!"))
50   T!=t+1 && throw(DomainError("Background score array
51     ↳ must have the same observation lengths as
52     ↳ observations!"))
53 end

54
55 function
56   ↳ init_logPWM_sources(prior_vector::AbstractVector{<:Union{<:AbstractF
57   ↳ source_length_limits::UnitRange{<:Integer}})
58   srcvec = Vector{Tuple{Matrix{Float64},Int64}}()
59   for prior in prior_vector
60     if
61       ↳ typeof(prior)<:AbstractVector{<:Dirichlet{<:AbstractF
62       if length(prior) <
63         ↳ source_length_limits[1]
64         length_dist=DiscreteNonParametric(
65           [source_length_limits ... ],
66
67             ↳ [PRIOR_LENGTH_MASS,ones(length(source_len
68             ↳ PWM_length=rand(length_dist)
69             elseif source_length_limits[1] <
70               ↳ length(prior) <
71               ↳ source_length_limits[end]
72               length_dist=DiscreteNonParametric(
73                 [source_length_limits ... ],
74                 [
75
76                   ↳ (ones(length(prior))-source_length_lim

```

```

63
64           → (ones(source_length_limits[end]-length
65           ])
66           )
67           PWM_length=rand(length_dist)
68     else
69       PWM_length=rand(source_length_limits)
70   end
71
72   if PWM_length>length(prior)
73
74       → prior_coord=rand(-(PWM_length-length(prior)) :
75   else
76
77       → prior_coord=rand(1:length(prior)-PWM_length+1)
78   end
79
80   PWM = zeros(PWM_length,4)
81
82   curr_pos=prior_coord
83   for pos in 1:PWM_length
84     curr_pos < 1 || curr_pos >
85     → length(prior) ?
86     → dirichlet=Dirichlet(ones(4)/4) :
87     → dirichlet=prior[curr_pos]
88     PWM[pos, :] = rand(dirichlet)
89   end
90
91   push!(srcvec, (log.(PWM), prior_coord))
92
93   → #push the source PWM to the source
94   → vector with the prior coord idx to
95   → allow drawing from the appropriate
96   → prior dirichlets on permuting source
97   → length
98
99   elseif prior=false
100      PWM_length=rand(source_length_limits)
101      PWM=zeros(PWM_length,4)
102      dirichlet=Dirichlet(ones(4)/4)
103      for pos in 1:PWM_length
104        PWM[pos,:]=rand(dirichlet)
105      end
106      push!(srcvec, (log.(PWM), 1))
107
108   else

```

```

95             throw(ArgumentError("Bad prior supplied
96                         ← for ICA_PWM_Model!"))
97         end
98     return srcvec
99 end
100
101 function
102     → init_mix_matrix(mix_prior::Tuple{BitMatrix,<:AbstractFloat},
103     → no_observations::Integer, no_sources::Integer)
104     inform, uninform=mix_prior
105     if size(inform,2) > 0
106         @assert size(inform,1)==no_observations &&
107             → size(inform,2)≤ no_sources "Bad informative
108             → mix prior dimensions!"
109     end
110     @assert 0.0 ≤ uninform ≤ 1.0 "Uninformative mix
111             → prior not between 0.0 and 1.0!"
112     mix_matrix = falses(no_observations, no_sources)
113     if size(inform,2)>0
114         mix_matrix[:,1:size(inform,2)]=inform
115     end
116     for index in
117         → CartesianIndices((1:no_observations,size(inform,2)+1:no_sources))
118         rand() ≤ uninform && (mix_matrix[index] = true)
119     end
120     return mix_matrix
121 end
122
123
124 function Base.show(io::IO, m::ICA_PWM_Model;
125     → nsrc::Integer=length(m.sources), progress=false)
126     nsrc == 0 && (nsrc=length(m.sources))
127     nsrc>length(m.sources) && (nsrc=length(m.sources))
128     nsrc=length(m.sources) ? (srcstr="All") : (srcstr="Top $nsrc")
129
130     printidxs, printsrcs, printfreqs=sort_sources(m,nsrc)
131
132     printstyled(io, "ICA PWM Model $(m.name) w/ logLi $(m.log_Li)\n",
133     → bold=true)
134     println(io, srcstr*" sources:")
135     for src in 1:nsrc
136         print(io, "$$(printidxs[src]), $(printfreqs[src]*100)%: ")
137         pwmstr_to_io(io, printsrcs[src])
138     end

```

```

129     println(io)
130   end
131   println(m.origin)
132
133   progress && return(nsrc+4)
134 end
135
136 function sort_sources(m, nsrc)
137   printidxs=Vector{Integer}(){}
138   printsrcs=Vector{Matrix{Float64}}{}
139   printfreqs=Vector{Float64}{}
140
141   freqs=vec(sum(m.mix_matrix,dims=1)); total=size(m.mix_matrix,1)
142   sortfreqs=sort(freqs,rev=true); sortidxs=sortperm(freqs,rev=true)
143   for srcidx in 1:nsrc
144     push!(printidxs, sortidxs[srcidx])
145     push!(printsrs, m.sources[sortidxs[srcidx]][1])
146     push!(printfreqs, sortfreqs[srcidx]/total)
147   end
148   return printidxs, printsrs, printfreqs
149 end

```

15.3.4 /src/IPM/IPM_likelihood.jl

```

1 #LIKELIHOOD SCORING FUNCS
2 function
3   ↳ IPM_likelihood(sources::AbstractVector{<:Tuple{<:AbstractMatrix{<:AbstractFloat},,
3   ↳ observations::AbstractMatrix{<:Integer},,
3   ↳ obs_lengths::AbstractVector{<:Integer},,
3   ↳ bg_scores::AbstractArray{<:AbstractFloat}, mix::BitMatrix,
3   ↳ revcomp::Bool=REVCOMP, returncache::Bool=false,
3   ↳ cache::AbstractVector{<:AbstractFloat}=zeros(0),
3   ↳ clean::AbstractVector{<:Bool}=Vector(false(size(observations)[2])))
3   source_wmls=[size(source[1])[1] for source in sources]
4   0 = size(bg_scores)[2]
5   source_stops=[obsl-wml+1 for wml in source_wmls, obsl in obs_lengths]
6   ↳ #stop scanning th source across the observation as the source
6   ↳ reaches the end
6   L=maximum(obs_lengths)+1
7
8   obs_src_idxs=mix_pull_idxs(mix) #get vectors of sources emitting in
8   ↳ each obs

```

```

9
10    revcomp ? (srcs=[cat(source[1],revcomp_pwm(source[1]),dims=3) for
→      source in sources]; motif_expectations = [((MOTIF_EXPECT/2)/obs1)
→      for obs1 in obs_lengths]; mat_dim=2) :
11      (srcs=[source[1] for
→      source in sources]; ; motif_expectations = [(MOTIF_EXPECT/obs1)
→      for obs1 in obs_lengths]; mat_dim=1) #setup appropriate reverse
→      complemented sources if necessary and set
→      log_motif_expectation-nMica has 0.5 per base for including the
→      reverse complement, 1 otherwise
12
13
14    lme_vec=zeros(length(sources))
15
16    obs_lhs=Vector{Vector{Float64}}() #setup likelihood vecs for threaded
→      operation-reassembled on return
17    nt=Threads.nthreads()
18    for t in 1:nt-1
19      push!(obs_lhs,zeros(Int(floor(0/nt))))
20    end
21    push!(obs_lhs, zeros(Int(floor(0/nt)+(0%nt))))
22
23    Threads.@threads for t in 1:nt
24      revcomp && (weavevec=zeros(3))
25      revcomp ? (score_mat=zeros(maximum(source_stops),2)) :
26        (score_mat=zeros(maximum(source_stops)))
27      opt = floor(0/nt) #obs per thread
28      score_matrices=Vector{typeof(score_mat)}(undef, length(sources))
→        #preallocate
29      osi_emitting=Vector{Int64}() #preallocate
30      lh_vec = zeros(L) #preallocated likelihood vector is one position
→        (0 initialiser) longer than the longest obs
31
32      for i in 1:Int(opt+(t==nt)*(0%nt))
33        o=Int(i+(t-1)*opt)
34        if clean[o]
35          obs_lhs[t][i]=cache[o]
36        else
37          obsl = obs_lengths[o]
38          oidxs=obs_src_ids[o]
39          mixwmls=source_wmls[oidxs]

            obs_cardinality = length(oidxs) #the more sources, the
→              greater the cardinality_penalty
40          if obs_cardinality > 0

```

```

40          revcomp ? score_sources_ds!(score_mat,
41              ← score_matrices, view(srcts,oidxs),
42              ← view(observations,:,o),
43              ← view(source_stops,oidxs,o)) :
44                  score_sources_ss!(score_mat, score_matrices,
45                      ← view(srcts,oidxs), view(observations,:,o),
46                      ← view(source_stops,oidxs,o)) #get scores for
47                      ← this observation
48
49
50
51
52          lme_vec=motif_expectations[o]
53          penalty_sum = sum(lme_vec[1:obs_cardinality])
54          penalty_sum > 1. && (penalty_sum=1.)
55          cardinality_penalty=log(1.0-penalty_sum)
56
57      else
58          cardinality_penalty=0.0
59      end
60
61
62          revcomp ? (obs_lhs[t][i]=weave_scores_ds!(weavevec,
63              ← lh_vec, obsl, view(bg_scores,:,:,o), score_matrices,
64              ← oidxs, mixwmls, log(motif_expectations[o])),
65              ← cardinality_penalty, osi_emitting)) :
66              (obs_lhs[t][i]=weave_scores_ss!(lh_vec, obsl,
67                  ← view(bg_scores,:,:,o), score_matrices, oidxs,
68                  ← mixwmls, log(motif_expectations[o]),
69                  ← cardinality_penalty, osi_emitting))
70
71
72          empty!(osi_emitting)
73
74
75      end
76  end
77
78
79      returnncache ? (return lps([lps(obs_lhs[t]) for t in 1:nt]),
80                      ← vcat(obs_lhs...)) : (return lps([lps(obs_lhs[t]) for t in 1:nt]))
81  end
82
83  @inline function mix_pull_idxs(A::AbstractArray{Bool})
84      n=count(A)
85      S=[Vector{Int64}() for o in 1:size(A,1)]
86      cnt=1
87      for (i,a) in pairs(A)
88          if a
89              push!(S[i[1]],i[2])

```

```

70             cnt+=1
71         end
72     end
73     return S
74 end
75
76 @inline function
77     revcomp_pwm(pwm::AbstractMatrix{<:AbstractFloat}) #in
78     → order to find a motif on the reverse strand, we scan
79     → the forward strand with the reverse complement of the
80     → pwm, reordered 3' to 5', so that eg. an PWM for an
81     → ATG motif would become one for a CAT motif
82     return pwm[end:-1:1,end:-1:1]
83 end
84
85 @inline function score_sources_ds!(score_mat,
86     → score_matrices::AbstractVector{<:AbstractMatrix{<:AbstractFloat}}}
87     → sources, observation::AbstractVector{<:Integer},
88     → source_stops)
89     for (s,source) in enumerate(sources)
90         for t in 1:source_stops[s]
91             for position in 1:size(source,1)
92                 score_loc = t+position-1 #score_loc is
93                 → the position of the obs to be scored
94                 → by PWM
95                 score_mat[t,1] +=
96                 → source[position,observation[score_loc],1]
97                 → #add the appropriate log PWM value
98                 → from the source to the score
99                 score_mat[t,2] +=
100                 → source[position,observation[score_loc],2]
101                 → #add the appropriate log PWM value
102                 → from the source to the score
103
104             end
105         end
106         score_matrices[s]=score_mat[1:source_stops[s],:]
107         → #copy score matrix to vector
108         score_mat.=0. #reset score matrix
109     end
110 end
111
112 end
113
114

```

```

95      @inline function score_sources_ss!(score_mat,
96          score_matrices::AbstractVector{<:AbstractArray{<:AbstractFloat}}|,
97          sources, observation::AbstractVector{<:Integer},
98          source_stops)
99          for (s,source) in enumerate(sources)
100             for t in 1:source_stops[s]
101                 for position in 1:size(source,1)
102                     score_loc = t+position-1 #score_loc is
103                         ← the position of the obs to be scored
104                         ← by PWM
105                     score_mat[t] +=
106                         ← source[position,observation[score_loc]]
107                         ← #add the appropriate log PWM value
108                         ← from the source to the score
109
110             end
111         end
112         score_matrices[s]=score_mat[1:source_stops[s]]
113             ← #copy score matrix to vector
114         score_mat.=0. #reset score matrix
115     end
116
117     end
118
119     @inline function weave_scores_ds!(weavevec, lh_vec,
120         obsl::Integer, bg_scores::SubArray,
121         score_mat::AbstractVector{<:AbstractMatrix{<:AbstractFloat}}|,
122         obs_source_indices::AbstractVector{<:Integer},
123         source_wmls::AbstractVector{<:Integer},
124         log_motif_expectation::AbstractFloat,
125         cardinality_penalty::AbstractFloat, osi_emitting)
126         for i in 2:obsl+1 #i=1 is ithe lh_vec initializing 0,
127             ← i=2 is the score of the first background position
128             ← (ie t=1)
129             t=i-1
130             score = lps(lh_vec[i-1], bg_scores[t],
131                         ← cardinality_penalty)
132
133             #logic: all observations are scored from t=wml to
134             ← the end of the obs-therefore check at each
135             ← position for new sources to add (indexed by
136             ← vector position to retrieve source wml and
137             ← score matrix)

```

```

115         if
116             → length(osi_emitting)<length(obs_source_indices)
117                 for n in 1:length(obs_source_indices)
118                     if !(n in osi_emitting)
119                         t ≥ source_wmls[n] &&
120                             → (push!(osi_emitting,n))
121             end
122         end
123
124         for n in osi_emitting
125             wml = source_wmls[n]
126             from_score = lh_vec[i-wml+1] #score at the
127                 → first position of the PWM
128             score_array = score_mat[n] #get the source
129                 → score matrix
130             score_idx = t - wml + 1 #translate t to
131                 → score_array idx for emission score
132             f_emit_score = score_array[score_idx,1]
133                 → #emission score at the last position of
134                 → the PWM
135             r_emit_score = score_array[score_idx,2]
136
137             weavevec *= score, lps(from_score,
138                 → f_emit_score, log_motif_expectation),
139                 → lps(from_score, r_emit_score,
140                 → log_motif_expectation)
141
142             score=logsumexp(weavevec)
143         end
144         lh_vec[i] = score
145     end
146     return lh_vec[obs1+1]
147 end
148
149 @inline function weave_scores_ss!(lh_vec, obs1::Integer,
150     → bg_scores::SubArray,
151     → score_mat::AbstractVector{<:AbstractVector{<:AbstractFloat}},
152     → obs_source_indices::AbstractVector{<:Integer},
153     → source_wmls::AbstractVector{<:Integer},
154     → log_motif_expectation::AbstractFloat,
155     → cardinality_penalty::AbstractFloat, osi_emitting)

```

```

141         for i in 2:obsl+1 #i=1 is ithe lh_vec initializing 0,
142             ↳ i=2 is the score of the first background position
143             ↳ (ie t=1)
144             t=i-1
145             score = lps(lh_vec[i-1], bg_scores[t],
146                         ↳ cardinality_penalty)
147             #logic: all observations are scored from t=wml to
148             ↳ the end of the obs-therefore check at each
149             ↳ position for new sources to add (indexed by
150             ↳ vector position to retrieve source wml and
151             ↳ score matrix)
152             if
153                 ↳ length(osi_emitting)<length(obs_source_indices)
154                 for n in 1:length(obs_source_indices)
155                     if !(n in osi_emitting)
156                         t ≥ source_wmls[n] &&
157                         ↳ (push!(osi_emitting,n))
158                     end
159                 end
160             end
161             for n in osi_emitting
162                 wml = source_wmls[n]
163                 from_score = lh_vec[i-wml+1] #score at the
164                 ↳ first position of the PWM
165                 score_array = score_mat[n] #get the source
166                 ↳ score matrix
167                 score_idx = t - wml + 1 #translate t to
168                 ↳ score_array idx for emission score
169                 f_emit_score = score_array[score_idx]
170                 ↳ #emission score at the last position of
171                 ↳ the PWM
172
173                 score=logaddexp(score, lps(from_score,
174                         ↳ f_emit_score, log_motif_expectation))
175             end
176             lh_vec[i] = score
177         end
178         return lh_vec[obsl+1]
179     end

```

15.3.5 /src/IPM/IPM_prior_utilities.jl

```

1 function read_fa_wms_tr(path::String)
2     wms=Vector{Matrix{Float64}}(){}
3     wm=zeros(1,4)
4     f=open(path)
5     for line in eachline(f)
6         prefix=line[1:2]
7         prefix == "01" && (wm=transpose([parse(Float64,i) for i in
8             split(line)[2:end]]))
9         prefix != "01" && prefix != "NA" && prefix != "PO" && prefix !=
10            "//" && (wm=vcat(wm, transpose([parse(Float64,i) for i in
11            split(line)[2:end]])))
12         prefix == "//" && push!(wms, wm)
13     end
14     return wms
15 end
16
17 #wm_samples are in decimal probability space, not log space
18 function assemble_source_priors(no_sources::Integer,
19     ←   wm_samples::AbstractVector{<:AbstractMatrix{<:AbstractFloat}},
20     ←   prior_wt::AbstractFloat=PRIOR_WT) #estimate a dirichlet prior on
21     ←   wm_samples inputs; if the number of samples is lower than the number
22     ←   of sources, return a false bool for init and permutation functions
23     source_priors = Vector{Union{Vector{Dirichlet{Float64}},Bool}}()
24     for source in 1:no_sources
25         if source ≤ length(wm_samples)
26             push!(source_priors,
27                 estimate_dirichlet_prior_on_wm(wm_samples[source],
28                     prior_wt))
29         else
30             push!(source_priors, false)
31         end
32     end
33     return source_priors
34 end
35
36
37 function
38     ←   estimate_dirichlet_prior_on_wm(wm::AbstractMatrix{<:AbstractFloat},
39     ←   wt::AbstractFloat=PRIOR_WT)
40     for i in 1:size(wm)[1]

```

```

29             !(isprobvec(wm[i,:])) && throw(DomainError("Bad
30                 weight vec supplied to
31                 estimate_dirichlet_prior_on_wm! $(wm[i,:])"))
32         end
33     prior = Vector{Dirichlet{Float64}}()
34     for position in 1:size(wm)[1]
35         normvec=wm[position,:]
36         zero_idxs=findall(isequal(0.),wm[position,:])
37         normvec[zero_idxs].+=10^-99
38         push!(prior, Dirichlet(normvec.*wt))
39     end
40     return prior
41 end
42
43 function cluster_mix_prior!(df::DataFrame,
44     ← wms::AbstractVector{<:AbstractMatrix{<:AbstractFloat}})
45     mix=falses(size(df,1),length(wms))
46     for (o, row) in enumerate(eachrow(df))
47         row.cluster ≠ 0 && (mix[o,row.cluster]=true)
48     end
49
50     represented_sources=unique(df.cluster)
51     return mix[:,represented_sources]
52 end
53
54 function infocenter_wms_trim(wm::AbstractMatrix{<:AbstractFloat},
55     ← trimsize::Integer)
56     !(size(wm,2)==4) && throw(DomainError("Bad wm! 2nd dimension should
57         ← be size 4"))
58     infovec=get_pwm_info(wm, logsw=false)
59     maxval, maxidx=findmax(infovec)
60     upstream_extension=Int(floor((trimsize-1)/2))
61     downstream_extension=Int(ceil((trimsize-1)/2))
62     1+upstream_extension+downstream_extension > size(wm,1) &&
63         ← throw(DomainError("Src too short for trim! $upstream_extension
64         ← $downstream_extension"))
65     return
66     ← wm[max(1,maxidx-upstream_extension):min(maxidx+downstream_extension,size(wm,1))]
67 end
68
69 function filter_priors(target_src_no::Integer, target_src_size::Integer,
70     ← prior_wms::AbstractVector{<:AbstractMatrix{<:AbstractFloat}},
71     ← prior_mix::BitMatrix)

```

```

62     wms=Vector{Matrix{Float64}}(undef, target_src_no)
63     freqsort_idxs=sortperm([sum(prior_mix[:,s]) for s in
64         1:length(prior_wms)])
65     for i in 1:target_src_no
66         target_src_idx=freqsort_idxs[i]
67         wms[i]=infocenter_wms_trim(prior_wms[target_src_idx],
68             → target_src_size)
69     end
70     return wms
71 end

71 function combine_filter_priors(target_src_no::Integer,
72     → target_src_size::Integer,
73     → prior_wms :: Tuple{<:AbstractVector{<:AbstractMatrix{<:AbstractFloat}},AbstractVect
74     → prior_mix:: Tuple{BitMatrix, BitMatrix})
75     wms=Vector{Matrix{Float64}}(undef, target_src_no)
76     cat_wms=vcat(prior_wms[1],prior_wms[2])
77     first_freq=[sum(prior_mix[1][:,s]) for s in 1:length(prior_wms[1])]
78     second_freq=[sum(prior_mix[2][:,s]) for s in 1:length(prior_wms[2])]
79     freqsort_idxs=sortperm(vcat(first_freq,second_freq))
80     for i in 1:target_src_no
81         target_src_idx=freqsort_idxs[i]
82         wms[i]=infocenter_wms_trim(cat_wms[target_src_idx],
83             → target_src_size)
84     end
85     return wms
86 end

```

15.3.6 /src/ensemble/IPM_Eensemle.jl

```

1 mutable struct IPM_Eensemle
2     path::String #ensemble models and popped-out posterior samples
3         → serialised here
4     models::Vector{Model_Record} #ensemble keeps paths to serialised
5         → models and their likelihood tuples rather than keeping the
6         → models in memory
7
8     contour::AbstractFloat#current log_Li[end]
9     log_Li::Vector{AbstractFloat} #likelihood of lowest-ranked model
10        → at iterate i
11     log_Xi::Vector{AbstractFloat} #amt of prior mass included in
12        → ensemble contour at Li

```

```

8      log_wi::Vector{AbstractFloat} #width of prior mass covered in
9          ↵ this iterate
10     log_Lawi::Vector{AbstractFloat} #evidentiary weight of the
11         ↵ iterate
12     log_Zi::Vector{AbstractFloat} #ensemble evidence
13     Hi::Vector{AbstractFloat} #ensemble information
14
15
16     source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:Abstract
17         ↵ #source pwm priors
18     mix_prior::Tuple{BitMatrix,AbstractFloat} #prior on %age of
19         ↵ observations that any given source contributes to
20
21     bg_scores::Matrix{AbstractFloat} #precalculated background HMM
22         ↵ scores, same dims as obs
23
24     sample_posterior::Bool
25     posterior_samples::Vector{Model_Record} #list of posterior sample
26         ↵ records
27
28
29 #####IPM_Elensemle FUNCTIONS
30 IPM_Elensemle(path::String, no_models::Integer,
31     ↵ source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
32     ↵ mix_prior::Tuple{BitMatrix,<:AbstractFloat},
33     ↵ bg_scores::AbstractMatrix{<:AbstractFloat},
34     ↵ obs::AbstractArray{<:Integer}, source_length_limits;
35     ↵ posterior_switch::Bool=false) =
36     IPM_Elensemle(
37         path,
38         assemble_IPMs(path, no_models, source_priors, mix_prior,
39             ↵ bg_scores, obs, source_length_limits) ... ,
40         [-Inf], #L0 = 0
41         [0], #ie exp(0) = all of the prior is covered
42         [-Inf], #w0 = 0
43         [-Inf], #Lwi0 = 0

```

```

38     [-1e300], #Z0 = 0
39     [0], #H0 = 0,
40     obs,
41     [findfirst(iszero,obs[:,o])-1 for o in 1:size(obs)[2]],
42     source_priors,
43     mix_prior,
44     bg_scores, #precalculated background score
45     posterior_switch,
46     Vector{String}(),
47     no_models+1,
48     IPM_likelihood(init_logPWM_sources(source_priors,
49                     ↳ source_length_limits), obs, [findfirst(iszero,obs[:,o])-1 for
50                     ↳ o in 1:size(obs)[2]], bg_scores,
51                     ↳ falses(size(obs)[2],length(source_priors)))))

50 IPM_Elensembler(worker_pool::AbstractVector{<:Integer}, path::String,
51   ↳ no_models::Integer,
52   ↳ source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFlo
53   ↳ mix_prior::Tuple{BitMatrix,<:AbstractFloat},
54   ↳ bg_scores::AbstractMatrix{<:AbstractFloat}, obs::Array{<:Integer},
55   ↳ source_length_limits; posterior_switch::Bool=false}) =
56 IPM_Elensembler(
57   ↳ path,
58   ↳ distributed_IPM_assembly(worker_pool, path, no_models,
59     ↳ source_priors, mix_prior, bg_scores, obs,
60     ↳ source_length_limits)...,
61   ↳ [-Inf], #L0 = 0
62   ↳ [0], #ie exp(0) = all of the prior is covered
63   ↳ [-Inf], #w0 = 0
64   ↳ [-Inf], #Liwi0 = 0
65   ↳ [-1e300], #Z0 = 0
66   ↳ [0], #H0 = 0,
67   ↳ obs,
68   ↳ [findfirst(iszero,obs[:,o])-1 for o in 1:size(obs)[2]],
69   ↳ source_priors,
70   ↳ mix_prior,
71   ↳ bg_scores, #precalculated background score
72   ↳ posterior_switch,
73   ↳ Vector{String}(),
74   ↳ no_models+1,
```

```

68     IPM_likelihood(init_logPWM_sources(source_priors,
69     ↵   source_length_limits), obs, [findfirst(iszero,obs[:,o])-1 for
70     ↵   o in 1:size(obs)[2]], bg_scores,
71     ↵   falses(size(obs)[2],length(source_priors))))
72
73
74     function assemble_IPMs(path::String, no_models::Integer,
75     ↵   source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat}},<:AbstractVector{<:AbstractFloat},<:AbstractMatrix{<:AbstractFloat}}}, mix_prior::Tuple{BitMatrix,<:AbstractFloat}, bg_scores::AbstractArray{<:AbstractFloat}, obs::AbstractArray{<:Integer}, source_length_limits::UnitRange{<:Integer})
76     ↵   ensemble_records = Vector{Model_Record}()
77     ↵   !isdir(path) && mkpath(path)
78
79     @assert size(obs)[2]==size(bg_scores)[2]
80
81     @showprogress 1 "Assembling IPM ensemble ... " for model_no in
82     ↵   1:no_models
83     ↵   model_path = string(path,'/',$model_no)
84     ↵   if !isfile(model_path)
85     ↵       model = ICA_PWM_Model(string(model_no),
86     ↵         source_priors, mix_prior, bg_scores, obs,
87     ↵         source_length_limits)
88     ↵       serialize(model_path, model) #save the model to
89     ↵         the ensemble directory
90     ↵       push!(ensemble_records,
91     ↵         Model_Record(model_path,model.log_Li))
92     ↵   else #interrupted assembly pick up from where we left off
93     ↵       model = deserialize(model_path)
94     ↵       push!(ensemble_records,
95     ↵         Model_Record(model_path,model.log_Li))
96     ↵   end
97
98     return ensemble_records, minimum([record.log_Li for record in
99     ↵   ensemble_records])
100 end

```

```

91 function distributed_IPM_assembly(worker_pool::Vector{Int64},
92   path::String, no_models::Integer,
93   source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat},  

94   mix_prior::Tuple{BitMatrix,<:AbstractFloat},  

95   bg_scores::AbstractArray{<:AbstractFloat},  

96   obs::AbstractArray{<:Integer},  

97   source_length_limits::UnitRange{<:Integer}})
98   ensemble_records = Vector{Model_Record}()  

99   !isdir(path) && mkpath(path)  

100  

101   @assert size(obs)[2]==size(bg_scores)[2]  

102  

103   model_chan=  

104     → RemoteChannel(()→Channel{ICA_PWM_Model}(length(worker_pool)))  

105   job_chan = RemoteChannel(()→Channel{Union{Tuple,Nothing}}(1))  

106     put!(job_chan,(source_priors, mix_prior, bg_scores, obs,  

107     → source_length_limits))  

108  

109   sequence_workers(worker_pool, worker_assemble, job_chan,  

110     → model_chan)  

111  

112   assembly_progress=Progress(no_models, desc="Assembling IPM  

113     → ensemble ...")  

114  

115   model_counter=check_assembly!(ensemble_records, path, no_models,  

116     → assembly_progress)  

117  

118   while model_counter ≤ no_models
119     wait(model_chan)
120     candidate=take!(model_chan)
121     model = ICA_PWM_Model(string(model_counter),
122       → candidate.origin, candidate.sources,
123       → candidate.source_length_limits, candidate.mix_matrix,
124       → candidate.log_Li)
125     model_path=string(path,'/',model_counter)
126     serialize(model_path,model)
127     push!(ensemble_records,
128       → Model_Record(model_path,model.log_Li))
129     model_counter+=1
130     next!(assembly_progress)
131   end
132  

133   take!(job_chan),put!(job_chan,nothing)

```

```

119
120     return ensemble_records, minimum([record.log_Li for record in
121         ↪ ensemble_records])
122 end
123
124     function
125         ↪ check_assembly!(ensemble_records::AbstractVector{
126             ↪ path::String, no_models::Integer,
127             ↪ assembly_progress::Progress)
128             counter=1
129             while counter ≤ no_models
130                 model_path=string(path,'/',counter)
131                 if isfile(model_path)
132                     model=deserialize(model_path)
133                     push!(ensemble_records,
134                         ↪ Model_Record(model_path, m))
135                     counter+=1
136                     next!(assembly_progress)
137             else
138                 return counter
139             end
140         end
141         return counter
142     end
143
144     function
145         ↪ worker_assemble(job_chan::RemoteChannel,
146             ↪ models_chan::RemoteChannel,
147             ↪ comms_chan::RemoteChannel)
148             put!(comms_chan,myid())
149             wait(job_chan)
150             params=fetch(job_chan)
151             while !(fetch(job_chan) ===
152                 ↪ nothing)
153                 model=ICA_PWM_Model(string(myid()),pa
154                 put!(models_chan,model)
155             end
156         end
157     end
158
159     function Base.show(io::IO, e::IPM_Engineering; nsrc=0, progress=false)
160         livec=[model.log_Li for model in e.models]
161         maxLH=maximum(livec)
162         printstyled(io, "ICA PWM Model Ensemble @ $(e.path)\n",
163             ↪ bold=true)

```

```

152 msg = @sprintf "Contour: %3.6e MaxLH:%3.3e Max/Naive:%3.3e log
153   ↳ Evidence:%3.6e" e.contour maxLH (maxLH-e.naive_lh)
154   ↳ e.log_Zi[end]
155   println(io, msg)
156   hist=UnicodePlots.histogram(livec, title="Ensemble Likelihood
157   ↳ Distribution")
158   show(io, hist)
159   println()
160   progress && return(nrows(hist.graphics)+6)
161 end

```

15.3.7 /src/ensemble/ensemble_utilities.jl

```

1 function ensemble_history(e::IPM_Ensemble, bins=25)
2     !e.sample_posterior && throw(ArgumentError("This ensemble has no
3         → posterior samples to show a history for!"))
4     livec=vcat([model.log_Li for model in e.models],[model.log_Li for
5         → model in e.posterior_samples])
6     show(histogram(livec, nbins=bins))
7 end
8
9 function e_backup(e::IPM_Ensemble, tuner::Permute_Tuner)
10    serialize(string(e.path,'/','ens'), e)
11    serialize(string(e.path,'/','tuner'), tuner)
12 end
13
14 function clean_ensemble_dir(e::IPM_Ensemble, model_pad::Integer;
15     → ignore_warn=false)
16     !ignore_warn && e.sample_posterior && throw(ArgumentError("Ensemble
17         → is set to retain posterior samples and its directory should not
18         → be cleaned!"))
19     for file in readdir(e.path)
20         !(file in vcat([basename(model.path) for model in
21             → e.models],"ens",[string(number) for number in
22                 → e.model_counter-length(e.models)-model_pad:e.model_counter-1]))
23             → && rm(e.path*'*file)
24     end
25 end
26
27 function complete_evidence(e::IPM_Ensemble)
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1166
1167
1168
1168
1169
1170
1171
1172
1173
1174
1175
1175
1176
1177
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1193
1194
1195
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1208
1209
1210
1211
1212
1213
1214
1215
1216
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1236
1237
1238
1238
1239
1240
1241
1242
1243
1244
1245
1245
1246
1247
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1256
1257
1258
1258
1259
1260
1261
1262
1263
1264
1265
1265
1266
1267
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1294
1295
1296
1296
1297
1298
1298
1299
1300
1301
1302
1303
1304
1304
1305
1306
1306
1307
1308
1308
1309
1310
1311
1312
1313
1314
1315
1315
1316
1317
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1325
1326
1327
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1335
1336
1337
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1345
1346
1347
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1355
1356
1357
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1365
1366
1367
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1375
1376
1377
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1394
1395
1396
1396
1397
1398
1398
1399
1400
1401
1402
1403
1403
1404
1405
1405
1406
1407
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1415
1416
1417
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1425
1426
1427
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1435
1436
1437
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1445
1446
1447
1447
1448
1449
1449
1450
1451
1452
1453
1454
1455
1455
1456
1457
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1465
1466
1467
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1475
1476
1477
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1494
1495
1496
1496
1497
1498
1498
1499
1500
1501
1502
1503
1503
1504
1505
1505
1506
1507
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1515
1516
1517
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1525
1526
1527
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1535
1536
1537
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1545
1546
1547
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1555
1556
1557
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1565
1566
1567
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1575
1576
1577
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1594
1595
1596
1596
1597
1598
1598
1599
1600
1601
1602
1603
1603
1604
1605
1605
1606
1607
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1615
1616
1617
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1625
1626
1627
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1635
1636
1637
1637
1638
1639
1639
1640
1641
1642
1643
1644
1645
1645
1646
1647
1647
1648
1649
1649
1650
1651
1652
1653
1654
1655
1655
1656
1657
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1665
1666
1667
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1675
1676
1677
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1694
1695
1696
1696
1697
1698
1698
1699
1700
1701
1702
1703
1703
1704
1705
1705
1706
1707
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1715
1716
1717
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1725
1726
1727
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1735
1736
1737
1737
1738
1739
1739
1740
1741
1742
1743
1744
1745
1745
1746
1747
1747
1748
1749
1749
1750
1751
1752
1753
1754
1755
1755
1756
1757
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1765
1766
1767
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1775
1776
1777
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1785
1786
1787
1787
1788
1789
1789
1790
1791
1792
1793
1794
1794
1795
1796
1796
1797
1798
1798
1799
1800
1801
1802
1803
1803
1804
1805
1805
1806
1807
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1815
1816
1817
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1825
1826
1827
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1835
1836
1837
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1845
1846
1847
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1855
1856
1857
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1865
1866
1867
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1875
1876
1877
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1885
1886
1887
1887
1888
1889
1889
1890
1891
1892
1893
1894
1894
1895
1896
1896
1897
1898
1898
1899
1900
1901
1902
1903
1903
1904
1905
1905
1906
1907
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1915
1916
1917
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1925
1926
1927
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1935
1936
1937
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1945
1946
1947
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1955
1956
1957
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1965
1966
1967
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1975
1976
1977
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1985
1986
1987
1987
1988
1989
1989
1990
1991
1992
1993
1994
1994
1995
1996
1996
1997
1998
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
23
```

```

20     return final_logZ = logaddexp(e.log_Zi[end], (logsumexp([model.log_Li
21       ↵ for model in e.models]) + e.log_Xi[length(e.log_Li)] -
22       ↵ log(length(e.models))))
23 end
24
25 function reset_ensemble!(e::IPM_Ensemble)
26   new_e=deepcopy(e)
27   for i in 1:length(e.models)
28     if string(i) in [basename(record.path) for record in e.models]
29       new_e.models[i]=e.models[findfirst(isequal(string(i)),
30         ↵ [basename(record.path) for record in e.models])]
31     else
32       new_e.models[i]=e.posterior_samples[findfirst(isequal(string(i)),
33         ↵ [basename(record.path) for record in
34           ↵ e.posterior_samples])]
35     end
36   end
37
38   new_e.contour=minimum([record.log_Li for record in new_e.models])
39
40   new_e.log_Li=[new_e.log_Li[1]]
41   new_e.log_Xi=[new_e.log_Xi[1]]
42   new_e.log_wi=[new_e.log_wi[1]]
43   new_e.log_Liwi=[new_e.log_Liwi[1]]
44   new_e.log_Zi=[new_e.log_Zi[1]]
45   new_e.Hi=[new_e.Hi[1]]
46
47   new_e.posterior_samples=Vector{Model_Record}()
48
49   new_e.model_counter=length(new_e.models)+1
50
51   clean_ensemble_dir(new_e, 0; ignore_warn=true)
52   isfile(e.path*"/tuner") && rm(e.path*"/tuner")
53   serialize(e.path*"/ens", new_e)
54
55   return new_e
56 end
57
58 function move_ensemble!(e::IPM_Ensemble,path::String)
59   !isdir(path) && mkdir(path)
60   for file in readdir(e.path)
61     mv(e.path*'*file,path*'*file)

```

```
57     end
58
59     for (n,model) in enumerate(e.models)
60         e.models[n]=Model_Record(path*'*basename(model.path),
61             ↪ model.log_Li)
62     end
63     if e.sample_posterior
64         for (n,model) in enumerate(e.posterior_samples)
65             ↪ e.posterior_samples[n]=Model_Record(path*'*basename(model.path),
66             ↪ model.log_Li)
67     end
68     rm(e.path)
69     e.path=path
70     serialize(e.path*"/ens",e)
71     return e
72 end
73
74 function copy_ensemble!(e::IPM_Ensemble,path::String)
75     new_e=deepcopy(e)
76     !isdir(path) && mkdir(path)
77     for file in readdir(e.path)
78         cp(e.path*'*file,path*'*file, force=true)
79     end
80
81     for (n,model) in enumerate(e.models)
82         new_e.models[n]=Model_Record(path*'*basename(model.path),
83             ↪ model.log_Li)
84     end
85     if e.sample_posterior
86         for (n,model) in enumerate(e.posterior_samples)
87             ↪ new_e.posterior_samples[n]=Model_Record(path*'*basename(model.path)
88             ↪ model.log_Li)
89     end
90     new_e.path=path
91     serialize(new_e.path*"/ens",e)
92     return new_e
93 end
```

```

94
95 function rewind_ensemble(e::IPM_Engsemble, rewind_idx)
96     !e.sample_posterior && throw(ArgumentError("An ensemble not retaining
97         ↪ posterior samples cannot be rewound!"))
98     rewind_idx ≥ length(e.log_Li) && throw(ArgumentError("rewind_idx
99         ↪ must be less than the current iterate!"))
100
101     n=length(e.models)
102     max_model_no=length(e.log_Li)+length(e.models)-1
103     rewind_model_no=rewind_idx+length(e.models)-1
104     new_e = deepcopy(e)
105
106     rm_models=[string(name) for name in rewind_model_no+1:max_model_no]
107
108     filter!(model→!(basename(model.path) in rm_models),new_e.models)
109     filter!(model→!(basename(model.path) in
110         ↪ rm_models),new_e.posterior_samples)
111
112     while length(new_e.models) < n
113         push!(new_e.models,pop!(new_e.posterior_samples))
114     end
115     new_e.contour=new_e.log_Li[rewind_idx]
116     new_e.log_Li=new_e.log_Li[1:rewind_idx]
117     new_e.log_Xi=new_e.log_Xi[1:rewind_idx]
118     new_e.log_wi=new_e.log_wi[1:rewind_idx]
119     new_e.log_Liwi=new_e.log_Liwi[1:rewind_idx]
120     new_e.log_Zi=new_e.log_Zi[1:rewind_idx]
121     new_e.Hi=new_e.Hi[1:rewind_idx]
122
123     new_e.model_counter=length(new_e.models)+rewind_idx
124
125     return new_e
126 end
127
128 function show_models(e::IPM_Engsemble, idxs)
129     liperm=sortperm([model.log_Li for model in e.models],rev=true)
130     for idx in idxs
131         m=deserialize(e.models[liperm[idx]].path)
132         show(m)
133     end
134 end
135
136 function get_model(e::IPM_Engsemble,no)

```

```

134     return deserialize(e.path*'*string(no))
135 end

```

15.3.8 /src/nested_sampler/converge_ensemble.jl

```

1 function converge_ensemble!(e::IPM_Ensemble,
2   instruction::Permute_Instruct; max_iterates=typemax(Int64),
3   backup::Tuple{Bool, Integer}=(false,0),
4   clean::Tuple{Bool, Integer, Integer}=(false,0,0), verbose::Bool=false,
5   converge_criterion::String="standard",
6   converge_factor::AbstractFloat=.001, progargs ... )
7   N = length(e.models); curr_it=length(e.log_Li)
8
9   curr_it>1 && isfile(e.path*"/tuner") ?
10    (tuner=deserialize(e.path*"/tuner")) : (tuner =
11      Permute_Tuner(instruction)) #restore tuner from saved if any
12   wk_mon = Worker_Monitor([1])
13   meter = ProgressNS(e, wk_mon, tuner, 0.; start_it=curr_it,
14    progargs ... )
15
16   converge_check = get_convfunc(converge_criterion)
17   while !converge_check(e, converge_factor) && (curr_it ≤
18     max_iterates)
19     warn,step_report = nested_step!(e, instruction) #step the
20     ensemble
21     warn == 1 && "#1" passed for warn code means no workers persist;
22     all have hit the permute limit
23     (@error "Failed to find new models, aborting at current
24       iterate."; return e) #if there is a warning, just
25     return the ensemble and print info
26   curr_it += 1
27
28   tune_weights!(tuner, step_report)
29   instruction = tuner.inst
30
31   backup[1] && curr_it%backup[2] == 0 && e_backup(e,tuner) #every
32     backup interval, serialise the ensemble and instruction
33   clean[1] && !e.sample_posterior && curr_it%clean[2] == 0 &&
34     clean_ensemble_dir(e,clean[3]) #every clean interval, remove
35     old discarded models
36
37   update!(meter, converge_check(e,converge_factor,vals=true) ... )

```

```

22     end
23
24     if converge_check(e,converge_factor)
25         final_logZ = complete_evidence(e)
26         @info "Job done, sampled to convergence. Final logZ $final_logZ"
27
28         e_backup(e,tuner)
29         clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final
30         ↵   clean
31         return final_logZ
32     elseif curr_it==max_iterates
33         @info "Job done, sampled to maximum iterate $max_iterates.
34             ↵   Convergence criterion not obtained."
35
36         e_backup(e,tuner)
37         clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final
38         ↵   clean
39         return e.log_Zi[end]
40     end
41
42 end
43
44 function converge_ensemble!(e::IPM_Ensemble,
45     ↵   instruction::Permute_Instruct, wk_pool::Vector{Int64};
46     ↵   max_iterates=typemax(Int64), backup::Tuple{Bool,Integer}=(false,0),
47     ↵   clean::Tuple{Bool,Integer,Integer}=(false,0,0), verbose::Bool=false,
48     ↵   converge_criterion::String="standard",
49     ↵   converge_factor::AbstractFloat=.001, progargs ... )
50     N = length(e.models)
51
52
53     converge_check = get_convfunc(converge_criterion)
54     model_chan=
55     ↵   RemoteChannel(()→Channel{Tuple{Union{ICA_PWM_Model,String},Integer,
56     ↵   AbstractVector{<:Tuple{}}}}(10*length(wk_pool))) #channel to take
57     ↵   EM iterates off of
58     job_chan =
59     ↵   RemoteChannel(()→Channel{Tuple{<:AbstractVector{<:Model_Record},
60     ↵   Float64, Union{Permute_Instruct,String}}}(1))
61     put!(job_chan,(e.models, e.contour, instruction))
62
63
64     if !converge_check(e,converge_factor) #sequence workers only if not
65     ↵   already converged
66     @async sequence_workers(wk_pool, permute_IPM, e, job_chan,
67     ↵   model_chan)

```

```

50    end
51
52    curr_it=length(e.log_Li)
53    wk_mon=Worker_Monitor(wk_pool)
54    curr_it>1 && isfile(e.path*/tuner) ?
55        (tuner=deserialize(e.path*/tuner)) : (tuner =
56            Permute_Tuner(instruction)) #restore tuner from saved if any
57    meter = ProgressNS(e, wk_mon, tuner, 0.; start_it=curr_it,
58        progargs ... )
59
60    while !converge_check(e, converge_factor) && (curr_it ≤
61        max_iterates)
62        warn, step_report = nested_step!(e, model_chan, wk_mon) #step the
63            ensemble
64        warn == 1 && #"1" passed for warn code means no workers persist;
65            all have hit the permute limit
66            (@error "All workers failed to find new models, aborting
67                at current iterate."; return e) #if there is a
68                warning, just return the ensemble and print info
69        curr_it += 1
70        tune_weights!(tuner, step_report)
71        instruction = tuner.inst
72        take!(job_chan); put!(job_chan,(e.models,e.contour,instruction))
73        backup[1] && curr_it%backup[2] == 0 && e_backup(e,tuner) #every
74            backup interval, serialise the ensemble and instruction
75        clean[1] && !e.sample_posterior && curr_it%clean[2] == 0 &&
76            clean_ensemble_dir(e,clean[3]) #every clean interval, remove
77            old discarded models
78
79        update!(meter, converge_check(e,converge_factor,vals=true) ... )
80    end
81
82    take!(job_chan); put!(job_chan, (e.models, e.contour, "stop")) #stop
83        instruction terminates worker functions
84
85    if converge_check(e,converge_factor)
86        final_logZ = complete_evidence(e)
87        @info "Job done, sampled to convergence. Final logZ $final_logZ"
88
89        e_backup(e,tuner)
90        clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final
91            clean
92        return final_logZ

```

```

80     elseif curr_it==max_iterates
81         @info "Job done, sampled to maximum iterate $max_iterates.
82             ↪ Convergence criterion not obtained."
83
84         e_backup(e,tuner)
85         clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final
86             ↪ clean
87         return e.log_Zi[end]
88     end
89
90     function evidence_converge(e, evidence_fraction;
91         → vals=false)
92         val=lps(findmax([model.log_Li for model in
93             → e.models])[1], e.log_Xi[end])
94         thresh=lps(log(evidence_fraction),e.log_Zi[end])
95         vals ? (return val, thresh) : (return val<thresh)
96     end
97
98     function compress_converge(e, compression_ratio;
99         → vals=false)
100        val=findmax([model.log_Li for model in
101            → e.models])[1]-e.contour
102        thresh=compression_ratio
103        vals ? (return val, thresh) : (return val<thresh)
104    end
105
106    function get_convfunc(criterion)
107        if criterion == "standard"
108            return evidence_converge
109        elseif criterion == "compression"
110            return compress_converge
111        else
112            throw(ArgumentError("Convergence criterion
113                → $criterion not supported! Try \"standard\" or
114                → \"compression\"."))
115        end
116    end

```

15.3.9 /src/nested_sampler/nested_step.jl

```

1 ##### IMPLEMENTATION OF JEFF SKILLINGS' NESTED SAMPLING ALGORITHM #####
2 function nested_step!(e::IPM_Ensemble, instruction::Permute_Instruct)
3     N = length(e.models) #number of sample models/particles on the
4         ↪ posterior surface
5     i = length(e.log_Li) #iterate number, index for last values
6     j = i+1 #index for newly pushed values
7
8     e.contour, least_likely_idx = findmin([model.log_Li for model in
9         ↪ e.models])
10    Li_model = e.models[least_likely_idx]
11
12    #SELECT NEW MODEL, SAVE TO ENSEMBLE DIRECTORY, CREATE RECORD AND PUSH
13        ↪ TO ENSEMBLE
14    model_selected=false; step_report=0
15    while !model_selected
16        candidate,step_report=permute_IPM(e, instruction)
17        if !(candidate==nothing)
18            if !(candidate.log_Li in [m.log_Li for m in e.models])
19                model_selected=true
20
21                new_model_record =
22                    ↪ Model_Record(string(e.path,'/',e.model_counter),
23                    ↪ candidate.log_Li);
24                e.sample_posterior && push!(e.posterior_samples,
25                    ↪ Li_model)#if sampling posterior, push the model
26                    ↪ record to the ensemble's posterior samples vector
27                deleteat!(e.models, least_likely_idx)
28                push!(e.models, new_model_record);
29
30                final_model=ICA_PWM_Model(string(e.model_counter),
31                    ↪ candidate.origin, candidate.sources,
32                    ↪ candidate.source_length_limits, candidate.mix_matrix,
33                    ↪ candidate.log_Li, candidate.permute_blacklist)
34                serialize(new_model_record.path, final_model)
35
36                e.model_counter +=1
37            end
38        else
39            push!(e.models, Li_model)
40            return 1, step_report
41        end

```



```

63         if (candidate.log_Li > e.contour) && !(candidate.log_Li in
64             [m.log_Li for m in e.models])
65             model_selected=true
66
66         new_model_record =
67             Model_Record(string(e.path,'/'),e.model_counter),
68             candidate.log_Li);
69         e.sample_posterior && push!(e.posterior_samples,
70             Li_model)#if sampling posterior, push the model
71             record to the ensemble's posterior samples vector
72         deleteat!(e.models, least_likely_idx)
73         push!(e.models, new_model_record);
74
74         final_model=ICA_PWM_Model(string(e.model_counter),
75             candidate.origin, candidate.sources,
76             candidate.source_length_limits, candidate.mix_matrix,
77             candidate.log_Li, candidate.permute_blacklist)
78         serialize(new_model_record.path, final_model)
79
79         e.model_counter +=1
80     end
81     update_worker_monitor!(wk_mon,wk,true)
82 else
83     update_worker_monitor!(wk_mon,wk,false)
84     !any(wk_mon.persist) && (return 1,step_report)
85 end
86
87 #UPDATE ENSEMBLE QUANTITIES
88 push!(e.log_Li, minimum([model.log_Li for model in e.models])) #log
89     likelihood of the least likely model - the current ensemble ll
90     contour at Xi
91 push!(e.log_Xi, -i/N) #log Xi - crude estimate of the iterate's
92     enclosed prior mass
93 push!(e.log_wi, lps(e.log_Xi[i], -((j+1)/N)-log(2))) #log width of
94     prior mass spanned by the last step-trapezoidal approx
95 push!(e.log_Liwi, lps(e.log_Li[j],e.log_wi[j])) #log likelihood + log
96     width = increment of evidence spanned by iterate
97 push!(e.log_Zi, logaddexp(e.log_Zi[i],e.log_Liwi[j]))    #log
98     evidence
99
100 #information- dimensionless quantity
101 push!(e.Hi, lps(

```

```

92         (exp(lps(e.log_Liwi[j], -e.log_Zi[j])) * e.log_Li[j]), #term1
93         (exp(lps(e.log_Zi[i], -e.log_Zi[j])) *
94             → lps(e.Hi[i], e.log_Zi[i])), #term2
95         -e.log_Zi[j])) #term3
96
97     return 0, step_report
98 end

```

15.3.10 /src/permuation/Permute_Tuner.jl

```

1 mutable struct Permute_Tuner
2     inst::Permute_Instruct
3     velocities::Matrix{Float64}
4     successes::BitMatrix #(memoryxfunc)
5     tabular_display::DataFrame
6     time_history::Vector{Float64}
7     override::Bool
8 end
9
10 """
11     Permute_Tuner(instruction)
12
13 Generate a Permute_Tuner for the given instruction.
14 """
15 function Permute_Tuner(instruction::Permute_Instruct)
16     nfuncs=length(instruction.funcs)
17     funcnames=Vector{String}()
18     vels=ones(TUNING_MEMORY*instruction.func_limit,nfuncs)
19     succs=true.(TUNING_MEMORY*instruction.func_limit,nfuncs)
20     for (idx,func) in enumerate(instruction.funcs)
21         length(instruction.args[idx])>0 ?
22             → (kwstr="+(length(instruction.args[idx]))kwa)") : (kwstr="")
23         push!(funcnames, string(nameof(func),kwstr))
24     end
25
26     tabular_display=DataFrame("Function"⇒funcnames,
27         → "Succeed"⇒zeros(Int64,nfuncs), "Fail"⇒zeros(Int64,
28         → nfuncs), "Velocity"⇒ones(nfuncs), "Weights"⇒instruction.weights)
29
30     instruction.override_time>0. ? (override=true) : (override=false)
31     return
32         → Permute_Tuner(instruction,vels,succs,tabular_display,zeros(CONVERGENCE_MEMORY)

```

```

29 end

30
31 """
32     tune_weights!(tuner, call_report)

33
34 Given a call_report from permute_IPM(), adjust tuner's Permute_Instruct
    ↳ weights for function success rate and likelihood surface velocity.
35 """
36 function tune_weights!(tuner::Permute_Tuner,
    ↳ call_report::Vector{Tuple{Int64,Float64,Float64}})
    for call in call_report
        funcidx,time,distance=call
        distance!==-Inf &&
            ↳ (tuner.velocities[:,funcidx]=update_velocity!(tuner.velocities[:,funcidx])
            ↳ #do not push velocity to array if it has -Inf probability
            ↳ (usu no new model found)
        if call==call_report[end]

41
42         ↳ tuner.successes[:,funcidx]=update_sucvec!(tuner.successes[:,funcidx],
43             tuner.tabular_display[funcidx,"Succeed"]+=1
44     else
45
46         ↳ tuner.successes[:,funcidx]=update_sucvec!(tuner.successes[:,funcidx],
47             tuner.tabular_display[funcidx,"Fail"]+=1
48     end
49     end
50 end

51
52 function update_velocity!(velvec,time,distance)
53     popfirst!(velvec) #remove first value
54     vel = distance - log(time)
55     push!(velvec,distance-log(time))
56 end

57
58 function update_sucvec!(sucvec, bool)
59     popfirst!(sucvec)
60     push!(sucvec, bool)
61 end

62

```

```

63 function update_weights!(t::Permute_Tuner)
64     mvels=[mean(t.velocities[:,n]) for n in 1:length(t.inst.funcs)]
65     t.tabular_display[!, "Velocity"]=copy(mvels)
66
67     any(i→i<0,mvels) && (mvels.+=-minimum(mvels)+1.) #to calculate
       ↳ weights, scale negative values into >1.
68     pvec=[mvels[n]*(sum(t.successes[:,n])/length(t.successes[:,n])) for n
       ↳ in 1:length(t.inst.funcs)]
69     pvec./=sum(pvec)
70     (any(pvec.<t.inst.min_clmps) || any(pvec.>t.inst.max_clmps)) &&
       ↳ clamp_pvec!(pvec,t.inst.min_clmps,t.inst.max_clmps)
71
72     @assert isprobvec(pvec)
73     t.inst.weights=pvec; t.tabular_display[!, "Weights"]=pvec
74 end
75
76 """
77     clamp_pvec(pvec, tuner)
78
79 Clamp the values of a probability vector between the minimums and
       ↳ maximums provided by a Permute_Tuner.
80 """
81         function clamp_pvec!(pvec, min_clmps, max_clmps)
82             #logic- first accumulate on low values, then distribute
               ↳ excess from high values
83             any(pvec.<min_clmps) ? (low_clamped=false) :
               ↳ (low_clamped=true)
84             while !low_clamped
85                 vals_to_accumulate=pvec.<min_clmps
86                 vals_to_deplete=pvec.>min_clmps
87
88                 ↳ depletion=sum(min_clmps[vals_to_accumulate]--pvec[vals_to_accumulate])
89
               ↳ pvec[vals_to_accumulate]=min_clmps[vals_to_accumulate]
90
               ↳ pvec[vals_to_deplete]=depletion/sum(vals_to_deplete)
91
92             !any(pvec.<min_clmps)&&(low_clamped=true)
93         end
94
95         any(pvec.>max_clmps) ? (high_clamped=false) :
               ↳ (high_clamped=true)

```

```

96         while !high_clamped
97             vals_to_deplete=pvec.>max_clmps
98             vals_to_accumulate=pvec.<max_clmps
99
100            → depletion=sum(pvec[vals_to_deplete]--max_clmps[vals_to_deplete])
101            pvec[vals_to_deplete]=max_clmps[vals_to_deplete]
102            → pvec[vals_to_accumulate].+=depletion/sum(vals_to_accumulate)
103
104            !any(pvec.>max_clmps) && (high_clamped=true)
105        end
106    end
107
108 function Base.show(io::IO, tuner::Permute_Tuner; progress=false)
109     show(io, tuner.tabular_display, rowlabel=:I, summary=false)
110     progress && return(size(tuner.tabular_display,1)+4)
111 end

```

15.3.11 /src/permuation/orthogonality_helper.jl

```

1 ##ORTHOGONALITY_HELPER
2 function consolidate_srcs(con_idxs::Dict{Integer,Vector{Integer}}|,
3     → m::ICA_PWM_Model, obs_array::AbstractMatrix{<:Integer},
4     → obs_lengths::AbstractVector{<:Integer},
5     → bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
6     → models::AbstractVector{<:Model_Record};
7     → iterates::Integer=length(m.sources)*2, remote=false)
8     new_log_Li=-Inf; iterate = 1
9     T,0 = size(obs_array); T=T-1; S = length(m.sources)
10    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
11    a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths,
12        → bg_scores, new_mix, true, true)
13
14    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce
15        → a model more likely than the lh contour or exceed iterates
16        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
17        clean=Vector{Bool}(trues(0))
18
19        for host_src in filter(!in(vcat(values(con_idxs)...)), keys(con_idxs)) #copy mix information to the source to be
20            → consolidated on as host

```

```

13     for cons_src in con_idxs[host_src]
14         new_mix[:,host_src] = [new_mix[o,host_src] ||
15             new_mix[o,cons_src] for o in 1:size(new_mix,1)]
16     end
17
18     remote ? (merger_m = deserialize(rand(models).path)) : (merger_m
19         = remotecall_fetch(deserialize, 1, rand(models).path))
20         #randomly select a model to merge
21 used_m_srcs=Vector{Int64}()
22
23 for src in unique(vcat(values(con_idxs)...)) #replace all
24     non-host sources with sources from a merger model unlike the
25     one being removed
26     distvec=[pwm_distance(m.sources[src][1],m_src[1]) for m_src
27         in merger_m.sources]
28     m_src=findmax(distvec)[2]
29     while m_src in used_m_srcs
30         distvec[m_src]=0.; m_src=findmax(distvec)[2]
31         length(used_m_srcs)=length(merger_m.sources) &&
32             break;break
33     end
34
35     clean[new_mix[:,src]]=false #mark dirty any obs that start
36         with the source
37     new_sources[src]=merger_m.sources[m_src]
38     new_mix[:,src]=merger_m.mix_matrix[:,m_src]
39     clean[new_mix[:,src]]=false #mark dirty any obs that end
40         with the source
41     push!(used_m_srcs, m_src)
42 end
43
44 if consolidate_check(new_sources)[1] #if the new sources pass the
45     consolidate check
46     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
47         obs_lengths, bg_scores, new_mix, true, true, cache,
48         clean) #assess likelihood
49 end
50
51 iterate += 1
52 end

```

```

43     return ICA_PWM_Model("candidate", "consolidated $(m.origin)",
44         → new_sources, m.source_length_limits, new_mix, new_log_Li)
45 end
46
47 function
48     → consolidate_check(sources :: AbstractVector{<:Tuple{<:AbstractMatrix{<:AbstractFloa
49     → thresh=CONSOLIDATE_THRESH, revcomp=REVCOMP)
50     pass=true
51     lengthδmat=[size(src1[1],1) - size(src2[1],1) for src1 in sources,
52         → src2 in sources]
53     cons_idxs=Dict{Integer,Vector{Integer}}()
54     for src1 in 1:length(sources), src2 in src1+1:length(sources)
55         if lengthδmat[src1,src2]==0
56             revcomp ? (info_condition =
57                 → pwm_distance(sources[src1][1],sources[src2][1]) < thresh
58                 ||
59                 → pwm_distance(sources[src1][1],revcomp_pwm(sources[src2][1]))
60                 < thresh)) : (info_condition =
61                 → (pwm_distance(sources[src1][1],sources[src2][1]) <
62                     → thresh))
63             if info_condition
64                 if !in(src1,keys(cons_idxs))
65                     cons_idxs[src1]=[src2]; pass=false
66                 else
67                     push!(cons_idxs[src1], src2)
68                 end
69             end
70         end
71     end
72
73     return pass, cons_idxs
74 end
75
76 function pwm_distance(pwm1,pwm2)
77     minwml=min(size(pwm1,1),size(pwm2,1))
78     return sum([euclidean(exp.(pwm1[pos,:]),
79         → exp.(pwm2[pos,:])) for pos in 1:minwml])/minwml
80 end

```

15.3.12 /src/permuation/permute_control.jl

```

1 mutable struct Permute_Instruct
2     funcs :: AbstractVector{<:Function}
3     weights :: AbstractVector{<:AbstractFloat}
4     args :: AbstractVector{<:AbstractVector{<:Tuple{<:Symbol,<:Any}}}
5     model_limit :: Integer
6     func_limit :: Integer
7     min_clmps :: AbstractVector{<:AbstractFloat}
8     max_clmps :: AbstractVector{<:AbstractFloat}
9     override_time :: AbstractFloat
10    override_weights :: AbstractVector{<:AbstractFloat}
11    Permute_Instruct(funcs,
12                      weights,
13                      model_limit,
14                      func_limit;
15                      min_clmps=fill(.01,length(funcs)),
16                      max_clmps=fill(1.,length(funcs)),
17                      override_time=0.,
18                      override_weights=zeros(length(funcs)),
19                      args=[Vector{Tuple{Symbol,Any}}() for i in
19                         1:length(funcs)])=assert_permute_instruct(funcs,weights,args,
20                         &&
21                         new(funcs,weights,args,model_limit,func_limit,min_clmps,max_c
22 end
23
24 function
25     ← assert_permute_instruct(funcs,weights,args,model_limit,func_limit,min_clmps,
26     ← max_clmps, override_time, override_weights)
27
28     ← !(length(funcs)==length(args)==length(weights)==length(min_clmps)==length(max_
29     ← && throw(ArgumentError("A valid Permute_Instruct must have as
30     ← many tuning weights and argument vectors as functions!")))
31     model_limit<1 && throw(ArgumentError("Permute_Instruct limit on
32     ← models to permute must be positive Integer!"))
33     func_limit<1 && throw(ArgumentError("Permute_Instruct limit on
34     ← fuction calls per model permuted must be positive Integer!"))
35     any(min_clmps.≥max_clmps) && throw(ArgumentError("Permute_Instruct
36     ← max_clmps must all be greater than corresponding min_clmps!"))
37     sum(min_clmps)>1. && throw(ArgumentError("Sum of minimum clamps must
38     ← be <1.0 to maintain a valid probability vector!"))
39     any(max_clmps.>1.) && throw(ArgumentError("Permute_Tuner maximum
40     ← clamps cannot be >1.0!"))

```

```

29     override_time<0. && throw(ArgumentError("Permute_Instruct
    ↵   override_time must be 0 (disabled) or positive float!"))
30     override_time>0. && !isprobvec	override_weights) &&
    ↵   throw(ArgumentError("Permute_Instruct override_weights must be
    ↵   valid probvec!"))
31     return true
32 end
33
34
35 #permutation routine function-
36 #general logic: receive array of permutation parameters, until a model
    ↵   more likely than the least is found:
37 #randomly select a model from the ensemble (the least likely having been
    ↵   removed by this point), then sample new models by permuting with each
    ↵   of hte given parameter sets until a model more likely than the
    ↵   current contour is found
38 #if none is found for the candidate model, move on to another candidate
    ↵   until the models_to_permute iterate is reached, after which return
    ↵   nothing for an error code
39
40 #four permutation modes: source (iterative random changes to sources
    ↵   until model lh>contour or iterate limit reached)
41 #                                     -(iterates, weight shift
    ↵   freq per source base, length change freq per source,
    ↵   weight_shift_dist (a ContinuousUnivariateDistribution)) for permute
    ↵   params
42 #                                     mix (iterative random
    ↵   changes to mix matrix as above)
43 #                                     -(iterates, unitrange of
    ↵   # of moves)
44 #                                     init (iteratively
    ↵   reinitialize sources from priors)
45 #                                     -(iterates) for init
    ↵   params
46 #                                     merge (iteratively copy
    ↵   a source + mix matrix row from another model in the ensemble until
    ↵   lh>contour or
    ↵   iterate
    ↵                                     limit reached)
47 #                                     -(iterates) for merge
    ↵   params
48
49
50 function permute_IPM(e::IPM_Ensemble, instruction::Permute_Instruct)

```

```

51     call_report=Vector{Tuple{Int64,Float64,Float64}}}()
52     for model = 1:instruction.model_limit
53         m_record = rand(e.models)
54         m = deserialize(m_record.path)
55
56         model_blacklist=copy(m.permute_blacklist)
57         filteridxs, filtered_funcs, filtered_weights, filtered_args =
58             ↳ filter_permutes(instruction, model_blacklist)
59
60         for call in 1:instruction.func_limit
61             start=time()
62             funcidx=rand(filtered_weights)
63             permute_func=filtered_funcs[funcidx]
64
65                 ↳ pos_args,kw_args=get_permfunc_args(permute_func,e,m,filtered_args[fun
66             new_m=permute_func(pos_args ... ;kw_args ... )
67
68                 ↳ push!(call_report,(filteridxs[funcidx],time()-start,new_m.log_Li
69                 ↳ - e.contour))
70
71             if length(new_m.permute_blacklist) > 0 && new_m.log_Li ==
72                 ↳ -Inf #in this case the blacklisted function will not work
73                 ↳ on this model at all; it should be removed from the
74                 ↳ functions to use
75                 vcat(model_blacklist,new_m.permute_blacklist)
76                 filteridxs, filtered_funcs, filtered_weights,
77                 ↳ filtered_args = filter_permutes(instruction,
78                 ↳ model_blacklist)
79             end
80
81             dupecheck(new_m,m) && new_m.log_Li > e.contour &&
82                 ↳ return new_m, call_report
83             end
84         end
85         return nothing, call_report
86     end
87
88     function permute_IPM(e::IPM_Ensemble, job_chan::RemoteChannel,
89         ↳ models_chan::RemoteChannel, comms_chan::RemoteChannel)
90         ↳ #ensemble.models is partially updated on the worker to populate
91         ↳ arguments for permute funcs
92             persist=true
93             id=myid()

```

```

81     put!(comms_chan,id)
82     while persist
83         wait(job_chan)
84         start=time()
85         e.models, e.contour, instruction = fetch(job_chan)
86         instruction = "stop" && (persist=false; break)
87
88         call_report=Vector{Tuple{Int64,Float64,Float64}}()
89         for model=1:instruction.model_limit
90             found::Bool=false
91             m_record = rand(e.models)
92
93             remotecall_fetch(isfile,1,m_record.path) ? m =
94             → (remotecall_fetch(deserialize,1,m_record.path)) : (break)
95
96             model_blacklist=copy(m.permute_blacklist)
97             filteridxs, filtered_funcs, filtered_weights, filtered_args =
98             → filter_permutes(instruction, model_blacklist)
99
100            for call in 1:instruction.func_limit
101                start=time()
102                funcidx=rand(filtered_weights)
103                permute_func=filtered_funcs[funcidx]
104
105                → pos_args,kw_args=get_permfunc_args(permute_func,e,m,filtered_args)
106                new_m=permute_func(pos_args ... ;kw_args ... )
107
108                → push!(call_report,(filteridxs[funcidx],time()-start,new_m.log_Li
109                → - e.contour))
110
111                if length(new_m.permute_blacklist) > 0 && new_m.log_Li =
112                    -Inf #in this case the blacklisted function will not
113                    work on this model at all; it should be removed from
114                    the functions to use
115                    vcat(model_blacklist,new_m.permute_blacklist)
116                    filteridxs, filtered_funcs, filtered_weights,
117                    → filtered_args = filter_permutes(instruction,
118                    → model_blacklist)
119            end
120
121            dupecheck(new_m,m) && new_m.log_Li > e.contour &&
122            → ((put!(models_chan, (new_m ,id, call_report)));
123            → found=true; break)

```

```

112
113         end
114     found=true && break;
115     wait(job_chan)
116     fetch(job_chan)≠e.models && (break) #if the ensemble has
117     → changed during the search, update it
118     model=instruction.model_limit &&
119     → (put!(models_chan, ("quit", id,
120     → call_report));persist=false)#worker to put
121     → "quit" on channel if it fails to find a model
122     → more likely than contour
123
124     end
125   end
126
127   function filter_permutes(instruction, model_blacklist)
128
129     → filteridxs=findall(p→!in(p,model_blacklist),instruction.funcs)
130     filtered_funcs=instruction.funcs[filteridxs]
131     filtered_weights=filter_weights(instruction.weights,
132     → filteridxs)
133     filtered_args=instruction.args[filteridxs]
134     return filteridxs, filtered_funcs, filtered_weights,
135     → filtered_args
136   end
137
138
139   function filter_weights(weights, idxs)
140     deplete=0.
141     for (i, weight) in enumerate(weights)
142       !in(i, idxs) && (deplete+=weight)
143     end
144     weights[idxs].+=deplete/length(idxs)
145     new_weights=weights[idxs]./sum(weights[idxs])
146     return Categorical(new_weights)
147   end
148
149   function
150     → get_permfunc_args(func::Function,e::IPM_Ensemble,
151     → m::ICA_PWM_Model, args::Vector{Tuple{Symbol,Any}})
152     pos_args=[]
153     argparts=Base.arg_decl_parts(methods(func).ms[1])
154     argnames=[Symbol(argparts[2][n][1]) for n in
155     → 2:length(argparts[2])]
```

```

143         for argname in argnames #assemble basic positional
144             ← arguments from ensemble and model fields
145             if argname == Symbol('m')
146                 push!(pos_args,m)
147             elseif argname in fieldnames(IPM_Engsemble)
148                 push!(pos_args,getfield(e,argname))
149             elseif argname in fieldnames(ICA_PWM_Model)
150                 push!(pos_args,getfield(m,argname))
151             else
152                 throw(ArgumentError("Positional argument
153                             ← $argname of $func not available in the
154                             ← ensemble or model!"))
155             end
156         end
157
158         kw_args = NamedTuple()
159         if length(args) > 0 #if there are any keyword
160             ← arguments to pass
161             sym=Vector{Symbol}()
162             val=Vector{Any}()
163             for arg in args
164                 push!(sym, arg[1]); push!(val, arg[2])
165             end
166             kw_args = (;zip(sym,val)... )
167         end
168
169         return pos_args, kw_args
170     end
171
172     function dupecheck(new_model, model)
173         (new_model.sources==model.sources &&
174          ← new_model.mix_matrix==model.mix_matrix) ? (return
175          ← false) : (return true)
176     end

```

15.3.13 /src/permuation/permute_functions.jl

```

1 #DECORRELATION SEARCH PATTERNS
2
3
```

```

4 function permute_source(m::ICA_PWM_Model, models::Vector{Model_Record},
5   obs_array :: AbstractMatrix{<:Integer},
6   obs_lengths :: AbstractVector{<:Integer},
7   bg_scores :: AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
8   source_priors :: AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat}},<:Integer}}, iterates::Integer=length(m.sources)*2,
9   weight_shift_freq :: AbstractFloat=PWM_SHIFT_FREQ,
10  weight_shift_dist :: Distributions.ContinuousUnivariateDistribution=PWM_SHIFT_DIST,
11  length_change_freq :: AbstractFloat=PWM_LENGTHPERM_FREQ,
12  length_perm_range :: UnitRange{<:Integer}=LENGTHPERM_RANGE,
13  remote=false)
14 #weight_shift_dist is given in decimal probability values- converted to
15 # log space in permute_source_lengths!
16 new_log_Li=-Inf; iterate = 1
17 0 = size(obs_array,2);S = length(m.sources)
18 new_sources=deepcopy(m.sources);
19
20 a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
21   bg_scores, m.mix_matrix, true, true)
22
23 while new_log_Li <= contour && iterate <= iterates #until we produce
24   a model more likely than the lh contour or exceed iterates
25   new_sources=deepcopy(m.sources);
26   s = rand(1:S)
27   clean=Vector{Bool}(trues(0))
28   clean[m.mix_matrix[:,s]]=false #all obs with source are dirty
29
30   weight_shift_freq > 0 &&
31     (new_sources[s]=permute_source_weights(new_sources[s],
32       weight_shift_freq, weight_shift_dist))
33   rand() < length_change_freq &&
34     (new_sources[s]=permute_source_length(new_sources[s],
35       source_priors[s], m.source_length_limits, length_perm_range))
36
37   new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
38     obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
39     clean)
40   iterate += 1
41 end
42
43 cons_check, cons_idxs = consolidate_check(new_sources)

```

```

26     cons_check ? (return ICA_PWM_Model("candidate", "PS from $(m.name)",
27     ↵ new_sources, m.source_length_limits, m.mix_matrix, new_log_Li)) :
28     ↵ (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate",
29     ↵ "PS from $(m.name)", new_sources, m.source_length_limits,
30     ↵ m.mix_matrix, new_log_Li), obs_array, obs_lengths, bg_scores,
31     ↵ contour, models; remote=remote))
32 end
33
34 function permute_mix(m::ICA_PWM_Model,
35   ↵ obs_array::AbstractMatrix{<:Integer},
36   ↵ obs_lengths::AbstractVector{<:Integer},
37   ↵ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
38   ↵ iterates::Integer=10,
39   ↵ mix_move_range::UnitRange=1:length(m.mix_matrix), remote=false)
40   new_log_Li=-Inf; iterate = 1; 0 = size(obs_array,2);
41   new_mix=falses(size(m.mix_matrix))
42
43   a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
44     ↵ bg_scores, m.mix_matrix, true, true)
45   dirty=false
46
47   while (new_log_Li ≤ contour || !dirty) && iterate ≤ iterates #until
48     ↵ we produce a model more likely than the lh contour or exceed
49     ↵ iterates
50     mix_moves=rand(mix_move_range)
51     mix_moves > length(m.mix_matrix) && (mix_moves =
52       ↵ length(m.mix_matrix))
53
54     new_mix, clean = mix_matrix_decorrelate(m.mix_matrix, mix_moves)
55     ↵ #generate a decorrelated candidate mix
56     c_log_li, c_cache = IPM_likelihood(m.sources, obs_array,
57       ↵ obs_lengths, bg_scores, new_mix, true, true, cache, clean)
58     ↵ #calculate the model with the candidate mix
59     positive_indices=c_cache.>(cache) #obtain any obs indices that
60     ↵ have greater probability than we started with
61     clean=Vector{Bool}(trues(0)-positive_indices)
62     if any(positive_indices) #if there are any such indices
63       new_log_Li, cache = IPM_likelihood(m.sources, obs_array,
64         ↵ obs_lengths, bg_scores, new_mix, true, true, cache,
65         ↵ clean) #calculate the new model
66       dirty=true
67     end
68     iterate += 1

```

```

49   end
50
51   return ICA_PWM_Model("candidate", "PM from $(m.name)", m.sources,
52     ↳ m.source_length_limits, new_mix, new_log_Li, Vector{Function}())
53   ↳ #no consolidate check is necessary as sources havent changed
54 end
55
56 function perm_src_fit_mix(m::ICA_PWM_Model,
57   ↳ models::Vector{Model_Record}, obs_array::AbstractMatrix{<:Integer},
58   ↳ obs_lengths::AbstractVector{<:Integer},
59   ↳ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
60   ↳ source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFlo
61   ↳ iterates::Integer=length(m.sources)*2,
62   ↳ weight_shift_freq::AbstractFloat=PWM_SHIFT_FREQ,
63   ↳ length_change_freq::AbstractFloat=PWM_LENGTHPERM_FREQ,
64   ↳ length_perm_range::UnitRange{<:Integer}=LENGTHPERM_RANGE, weight_shift_dist::Distr
65   ↳ remote=false)
66   new_log_Li=-Inf; iterate = 1
67   O = size(obs_array,2); S = length(m.sources)
68   new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
69   fit_mix in m.permute_blacklist ? (new_bl=m.permute_blacklist) :
70   ↳ (new_bl=Vector{Function}())
71
72   while new_log_Li ≤ contour && iterate ≤ iterates #until we produce
73     ↳ a model more likely than the lh contour or exceed iterates
74     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix);
75     ↳ tm_one=deepcopy(m.mix_matrix);
76     clean=Vector{Bool}(trues(0))
77     s = rand(1:S);
78     clean[new_mix[:,s]]*=false #all obs starting with source are
79     ↳ dirty
80
81     new_mix[:,s]=false;tm_one[:,s]=true
82
83     weight_shift_freq > 0 &&
84     ↳ (new_sources[s]=permute_source_weights(new_sources[s],
85     ↳ weight_shift_freq, weight_shift_dist))
86     rand() < length_change_freq &&
87     ↳ (new_sources[s]=permute_source_length(new_sources[s],
88     ↳ source_priors[s], m.source_length_limits, length_perm_range))
89
90     l,zero_cache=IPM_likelihood(new_sources, obs_array, obs_lengths,
91     ↳ bg_scores, new_mix, true, true)

```

```

72     l,one_cache=IPM_likelihood(new_sources, obs_array, obs_lengths,
73         ↵ bg_scores, tm_one, true, true)
74     fit_mix=one_cache.≥zero_cache
75
76     if REVCOMP #if we're looking at reverse strands, we want to see
77         ↵ if sources fit better on the reverse strand, and if so switch
78         ↵ over to the revcomp source and use its fitted mix
79         revsrc=(revcomp_pwm(new_sources[s][1]),new_sources[s][2])
80         revsrcs=deepcopy(new_sources)
81         revsrcs[s]=revsrc
82         l, revsrc_cache=IPM_likelihood(revsrcs, obs_array,
83             ↵ obs_lengths, bg_scores, tm_one, true, true)
84         rfit_mix=revsrc_cache.≥zero_cache
85         lps(revsrc_cache[rfit_mix])>lps(one_cache[fit_mix]) &&
86             ↵ (new_sources=revsrcs;fit_mix=rfit_mix) #if the total
87             ↵ likelihood contribution for the revsource fit is greater
88             ↵ than the source fit, take the revsource
89     end
90
91     new_mix[:,s]=fit_mix
92
93     clean[fit_mix]∙∙=false #all obs ending with source are dirty
94
95     new_log_Li = IPM_likelihood(new_sources, obs_array, obs_lengths,
96         ↵ bg_scores, new_mix, true, false, zero_cache, clean)
97     iterate += 1
98
99     end
100
101     cons_check, cons_idxs = consolidate_check(new_sources)
102     cons_check ? (return ICA_PWM_Model("candidate","PSFM from
103         ↵ $(m.name)",new_sources, m.source_length_limits, new_mix,
104         ↵ new_log_Li, new_bl)) : (return consolidate_srcs(cons_idxs,
105         ↵ ICA_PWM_Model("candidate","PSFM from $(m.name)",new_sources,
106         ↵ m.source_length_limits, new_mix, new_log_Li, new_bl), obs_array,
107         ↵ obs_lengths, bg_scores, contour, models; remote=remote))
108 end
109
110 function fit_mix(m::ICA_PWM_Model, models::Vector{Model_Record},
111     ↵ obs_array::AbstractMatrix{<:Integer},
112     ↵ obs_lengths::AbstractVector{<:Integer},
113     ↵ bg_scores::AbstractMatrix{<:AbstractFloat}; remote=false)
114     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix);
115     ↵ test_mix=false(size(m.mix_matrix))

```

```

98     new_bh=[fit_mix]
99
100    l,zero_cache=IPM_likelihood(m.sources, obs_array, obs_lengths,
101      ↪ bg_scores, test_mix, true, true)
102    for (s,source) in enumerate(m.sources)
103      test_mix=falses(size(m.mix_matrix))
104      test_mix[:,s]=true
105      l,src_cache=IPM_likelihood(m.sources, obs_array, obs_lengths,
106        ↪ bg_scores, test_mix, true, true)
107      fit_mix=src_cache.≥zero_cache
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123

```

```

124 function random_decorrelate(m::ICA_PWM_Model,
125   → models :: Vector{Model_Record}, obs_array :: AbstractMatrix{<:Integer},
126   → obs_lengths :: AbstractVector{<:Integer},
127   → bg_scores :: AbstractMatrix{<:AbstractFloat}, contour :: AbstractFloat,
128   → source_priors :: AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat}},<:Integer}=length(m.sources)*2,
129   → source_permute_freq :: AbstractFloat=SRC_PERM_FREQ,
130   → weight_shift_freq :: AbstractFloat=PWM_SHIFT_FREQ,
131   → length_change_freq :: AbstractFloat=PWM_LENGTHPERM_FREQ,
132   → weight_shift_dist :: Distributions.ContinuousUnivariateDistribution=PWM_SHIFT_DIST,
133   → mix_move_range :: UnitRange=1:size(m.mix_matrix,1), remote=false)
134   new_log_Li=-Inf; iterate = 1
135   0 = size(obs_array,2); S = length(m.sources)
136   new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
137
138   a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths,
139   → bg_scores, new_mix, true, true)
140
141   while new_log_Li ≤ contour && iterate ≤ iterates #until we produce
142     → a model more likely than the lh contour or exceed iterates
143     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
144     clean=Vector{Bool}(trues(0))
145     s = rand(1:S)
146     clean[new_mix[:,s]]*=false #all obs starting with source are
147     → dirty
148     rand() < source_permute_freq &&
149       → (new_sources[s]=permute_source_weights(new_sources[s],
150           → weight_shift_freq, weight_shift_dist))
151     rand() < length_change_freq &&
152       → (new_sources[s]=permute_source_length(new_sources[s],
153           → source_priors[s], m.source_length_limits))
154
155     → new_mix[:,s]=mixvec_decorrelate(new_mix[:,s],rand(mix_move_range))
156     clean[new_mix[:,s]]*=false #all obs ending with source are dirty
157     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
158     → obs_lengths, bg_scores, new_mix, true, true, cache, clean)
159     iterate += 1
160
161 end
162
163
164 cons_check, cons_idxs = consolidate_check(new_sources)

```

```

145  cons_check ? (return ICA_PWM_Model("candidate", "RD from $(m.name)",
146    ↵ new_sources, m.source_length_limits, new_mix, new_log_Li)) :
147    ↵ (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate", "RD
148    ↵ from $(m.name)", new_sources, m.source_length_limits, new_mix,
149    ↵ new_log_Li), obs_array, obs_lengths, bg_scores, contour, models;
150    ↵ remote=remote))
151 end
152
153
154 function shuffle_sources(m::ICA_PWM_Model,
155   ↵ models::AbstractVector{<:Model_Record},
156   ↵ obs_array::AbstractMatrix{<:Integer},
157   ↵ obs_lengths::AbstractVector{<:Integer},
158   ↵ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
159   ↵ remote=false)
160   new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
161   new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
162
163   a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
164     ↵ bg_scores, m.mix_matrix, true, true)
165
166   remote ? (merger_m = deserialize(rand(models).path)) : (merger_m =
167     ↵ remotecall_fetch(deserialize, 1, rand(models).path))#randomly
168     ↵ select a model to merge
169
170   svec=[1:S ... ]
171
172   while new_log_Li <= contour && length(svec)>0 #until we produce a
173     ↵ model more likely than the lh contour or no more sources to
174     ↵ attempt merger
175     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
176     clean=Vector{Bool}(trues(0))
177
178     s = popat!(svec,rand(1:length(svec))) #randomly select a source
179       ↵ to merge
180
181     new_sources[s]=merger_m.sources[s];
182       ↵ new_mix[:,s] *= merger_m.mix_matrix[:,s] #shuffle in the merger
183       ↵ model source from this index
184
185     clean[m.mix_matrix[:,s]]*=false #mark dirty any obs that have the
186       ↵ source

```

```

167     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
168         ← obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
169         ← clean) #assess likelihood
170 end
171
172 cons_check, cons_idxs = consolidate_check(new_sources)
173 cons_check ? (return ICA_PWM_Model("candidate","SS from
174     ← $(m.name)",new_sources, m.source_length_limits, new_mix,
175     ← new_log_Li,Vector{Function}())) : (return
176     ← consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","SS from
177     ← $(m.name)",new_sources, m.source_length_limits, new_mix,
178     ← new_log_Li), obs_array, obs_lengths, bg_scores, contour, models;
179     ← remote=remote))
180
181 function accumulate_mix(m::ICA_PWM_Model,
182     ← models::AbstractVector{<:Model_Record},
183     ← obs_array::AbstractMatrix{<:Integer},
184     ← obs_lengths::AbstractVector{<:Integer},
185     ← bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
186     ← remote=false)
187     new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
188     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
189     new_bl=Vector{Function}()
190
191     a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
192         ← bg_scores, m.mix_matrix, true, true)
193
194     remote ? (merger_m = deserialize(rand(models).path)) : (merger_m =
195         ← remotecall_fetch(deserialize, 1, rand(models).path))#randomly
196         ← select a model to merge
197
198     svec=[1:S ... ]
199
200     while new_log_Li <= contour && length(svec)>0 #until we produce a
201         ← model more likely than the lh contour or no more sources to
202         ← attempt merger
203         new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
204         clean=Vector{Bool}(trues(O))
205
206         s = popat!(svec,rand(1:length(svec))) #randomly select a source
207             ← to merge

```

```

190     distvec=[pwm_distance(src[1],m_src[1]) for src in m.sources,
191     ↵   m_src in merger_m.sources]
192     S > 1 ? merge_s=findmin(distvec)[2][2] :
193     ↵   merge_s=findmin(distvec)[2]
194
195     new_mix[:,s]=[new_mix[o,s] || merger_m.mix_matrix[o,merge_s] for
196     ↵   o in 1:size(new_mix,1)] #accumulate the mix vector for this
197     ↵   source
198
199     clean[m.mix_matrix[:,s]]=false #mark dirty any obs that have the
200     ↵   source
201     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
202     ↵   obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
203     ↵   clean) #assess likelihood
204
205     if new_log_Li ≤ contour #if accumulating on the host source
206     ↵   doesnt work, try copying over the merger source
207     new_sources[s]=merger_m.sources[merge_s]
208     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
209     ↵   obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
210     ↵   clean) #assess likelihood
211   end
212 end
213
214 (new_log_Li = m.log_Li || new_log_Li in [model.log_Li for model in
215   ↵   models]) && (new_log_Li=-Inf) #accumulate can sometimes duplicate
216   ↵   models if source mix is identical
217
218 cons_check, cons_idxs = consolidate_check(new_sources)
219 cons_check ? (return ICA_PWM_Model("candidate","AM from
220   ↵   $(m.name)",new_sources, m.source_length_limits, new_mix,
221   ↵   new_log_Li)) : (return consolidate_srcs(cons_idxs,
222   ↵   ICA_PWM_Model("candidate","AM from $(m.name)",new_sources,
223   ↵   m.source_length_limits, new_mix, new_log_Li), obs_array,
224   ↵   obs_lengths, bg_scores, contour, models; remote=remote))
225
226 end
227
228 function distance_merge(m::ICA_PWM_Model,
229   ↵   models::AbstractVector{<:Model_Record},
230   ↵   obs_array::AbstractMatrix{<:Integer},
231   ↵   obs_lengths::AbstractVector{<:Integer},
232   ↵   bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
233   ↵   remote=false)

```

```

211     new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
212     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
213
214     a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths,
215     ↪ bg_scores, new_mix, true, true)
216
217     remote ? (merger_m = deserialize(rand(models).path)) : (merger_m =
218     ↪ remotecall_fetch(deserialize, 1, rand(models).path)) #randomly
219     ↪ select a model to merge
220
221     svec=[1:S ... ]
222
223
224     while new_log_Li <= contour && length(svec)>0 #until we produce a
225     ↪ model more likely than the lh contour or no more sources to
226     ↪ attempt merger
227     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
228     clean=Vector{Bool}(trues(0))
229
230     s = popat!(svec,rand(1:length(svec))) #randomly select a source
231     ↪ to merge
232     merge_s=most_dissimilar(new_mix[:,s],merger_m.mix_matrix) #find
233     ↪ the source in the merger model whose mixvec is most
234     ↪ dissimilar to the one selected
235
236     clean[new_mix[:,s]]-=false #mark dirty any obs that start with
237     ↪ the source
238     new_sources[s] = merger_m.sources[merge_s] #copy the source
239     new_mix[:,s] = merger_m.mix_matrix[:,merge_s] #copy the mixvector
240     ↪ (without which the source will likely be highly improbable)
241     clean[new_mix[:,s]]-=false #mark dirty any obs that end with the
242     ↪ source
243
244     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
245     ↪ obs_lengths, bg_scores, new_mix, true, true, cache, clean)
246     ↪ #assess likelihood
247   end
248
249   cons_check, cons_idxs = consolidate_check(new_sources)

```

```

236   cons_check ? (return ICA_PWM_Model("candidate","DM from
237   ↵ $(m.name)",new_sources, m.source_length_limits, new_mix,
238   ↵ new_log_Li,Vector{Function}())) : (return
239   ↵ consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","DM from
240   ↵ $(m.name)",new_sources, m.source_length_limits, new_mix,
241   ↵ new_log_Li), obs_array, obs_lengths, bg_scores, contour, models;
242   ↵ remote=remote))
243 end
244
245 function similarity_merge(m::ICA_PWM_Model,
246   ↵ models::AbstractVector{<:Model_Record},
247   ↵ obs_array::AbstractMatrix{<:Integer},
248   ↵ obs_lengths::AbstractVector{<:Integer},
249   ↵ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
250   ↵ remote=false)
251   new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
252   new_sources=deepcopy(m.sources)
253
254   a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
255   ↵ bg_scores, m.mix_matrix, true, true)
256
257   remote ? (merger_m = deserialize(rand(models).path)) : (merger_m =
258   ↵ remotecall_fetch(deserialize, 1, rand(models).path))#randomly
259   ↵ select a model to merge
260
261   svec=[1:S ... ]
262
263   while new_log_Li <= contour && length(svec)>0 #until we produce a
264   ↵ model more likely than the lh contour or no more sources to
265   ↵ attempt merger
266   new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
267   clean=Vector{Bool}(trues(O))
268
269   s = popat!(svec,rand(1:length(svec))) #randomly select a source
270   ↵ to merge
271   merge_s=most_similar(m.mix_matrix[:,s],merger_m.mix_matrix)
272   ↵ #obtain the source in the merger model whose mixvec is most
273   ↵ similar to the one in the original
274
275   clean[m.mix_matrix[:,s]]*=false #mark dirty any obs that have the
276   ↵ source
277   new_sources[s] = merger_m.sources[merge_s] #copy the source, but
278   ↵ dont copy the mixvector

```

```

258     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
259         ↵ obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
260         ↵ clean) #assess likelihood
261     end
262
263     cons_check, cons_idxs = consolidate_check(new_sources)
264     cons_check ? (return ICA_PWM_Model("candidate", "SM from
265         ↵ $(m.name)", new_sources, m.source_length_limits, m.mix_matrix,
266         ↵ new_log_Li, Vector{Function}())) : (return
267         consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate", "SM from
268         ↵ $(m.name)", new_sources, m.source_length_limits, m.mix_matrix,
269         ↵ new_log_Li), obs_array, obs_lengths, bg_scores, contour, models;
270         ↵ remote=remote))
271   end
272
273   function reinit_src(m::ICA_PWM_Model,
274     ↵ models::AbstractVector{<:Model_Record},
275     ↵ obs_array::AbstractMatrix{<:Integer},
276     ↵ obs_lengths::AbstractVector{<:Integer},
277     ↵ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
278     ↵ source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat}},<:Vector{Bool}}}},
279     ↵ iterates::Integer=length(m.sources)*2, remote=false)
280     new_log_Li=-Inf; iterate = 1
281     0 = size(obs_array,2); S = length(m.sources)
282     new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
283
284     while new_log_Li <= contour && iterate <= iterates #until we produce
285       ↵ a model more likely than the lh contour or exceed iterates
286       new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix);
287       ↵ tm_one=deepcopy(m.mix_matrix);
288       clean=Vector{Bool}(trues(0))
289       s = rand(1:S);
290       clean[new_mix[:,s]] = false #all obs starting with source are
291       ↵ dirty
292
293       new_mix[:,s] = false; tm_one[:,s] = true
294
295       new_sources[s] = init_logPWM_sources([source_priors[s]],
296         ↵ m.source_length_limits)[1] #reinitialize the source
297
298       l, zero_cache=IPM_likelihood(new_sources, obs_array, obs_lengths,
299         ↵ bg_scores, new_mix, true, true)

```

```

281     l,one_cache=IPM_likelihood(new_sources, obs_array, obs_lengths,
282     ↪ bg_scores, tm_one, true, true)
283
284     fit_mix=one_cache.≥zero_cache
285
286     new_mix[:,s]=fit_mix
287
288     clean[fit_mix]•=false #all obs ending with source are dirty
289
290     new_log_Li = IPM_likelihood(new_sources, obs_array, obs_lengths,
291     ↪ bg_scores, new_mix, true, false, zero_cache, clean)
292     iterate += 1
293   end
294
295   cons_check, cons_idxs = consolidate_check(new_sources)
296   cons_check ? (return ICA_PWM_Model("candidate","RS from $(m.name)",
297     ↪ new_sources, m.source_length_limits, new_mix,
298     ↪ new_log_Li,Vector{Function}())) : (return
299     ↪ consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","RS from
300     ↪ $(m.name)",new_sources, m.source_length_limits, new_mix,
301     ↪ new_log_Li), obs_array, obs_lengths, bg_scores, contour, models;
302     ↪ remote=remote))
303 end
304
305 function erode_model(m::ICA_PWM_Model,
306   ↪ models::AbstractVector{<:Model_Record},
307   ↪ obs_array::AbstractMatrix{<:Integer},
308   ↪ obs_lengths::AbstractVector{<:Integer},
309   ↪ bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
310   ↪ info_thresh::AbstractFloat=EROSION_INFO_THRESH, remote=false)
311   new_log_Li=-Inf; O = size(obs_array,2);
312   new_sources=deepcopy(m.sources)
313
314   erosion_sources=Vector{Integer}()
315   for (s,src) in enumerate(m.sources)
316     pwm,pi=src
317     if size(pwm,1)>m.source_length_limits[1] #do not consider
318       ↪ eroding srcs at min length limit
319       infovec=get_pwm_info(pwm)
320       any(info→<(info, info_thresh),infovec) &&
321       ↪ push!(erosion_sources,s)
322     end
323   end
324 end

```

```

309
310     length(erosion_sources)==0 && return ICA_PWM_Model("candidate","EM"
311         ←   from $(m.name)",new_sources, m.source_length_limits,
312         ←   m.mix_matrix, new_log_Li,[erode_model])#if we got a model we cant
313         ←   erode bail out with a model marked -Inf lh and with EM
314         ←   blacklisted
315
316     a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
317         ←   bg_scores, m.mix_matrix, true, true)
318
319     while new_log_Li ≤ contour && length(erosion_sources) > 0 #until we
320         ←   produce a model more likely than the lh contour or there are no
321         ←   more sources to erode
322         clean=Vector{Bool}(trues(0))
323         s=popat!(erosion_sources,rand(1:length(erosion_sources)))
324
325         new_sources[s]=erode_source(new_sources[s],
326             ←   m.source_length_limits, info_thresh)
327         clean[m.mix_matrix[:,s]]*=false
328
329         new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
330             ←   obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
331             ←   clean) #assess likelihood
332     end
333
334     new_log_Li ≤ contour ? (blacklist=[erode_model]) :
335         ←   (blacklist=Vector{Function}())
336
337     cons_check, cons_idxs = consolidate_check(new_sources)
338     cons_check ? (return ICA_PWM_Model("candidate","EM from
339         ←   $(m.name)",new_sources, m.source_length_limits, m.mix_matrix,
340         ←   new_log_Li, blacklist)) : (return consolidate_srcs(cons_idxs,
341         ←   ICA_PWM_Model("candidate","EM from $(m.name)",new_sources,
342         ←   m.source_length_limits, m.mix_matrix, new_log_Li, blacklist),
343         ←   obs_array, obs_lengths, bg_scores, contour, models;
344         ←   remote=remote))
345
346 end
347
348
349

```

```

330 function info_fill(m::ICA_PWM_Model,
331   → models :: AbstractVector{<:Model_Record},
332   → obs_array :: AbstractMatrix{<:Integer},
333   → obs_lengths :: AbstractVector{<:Integer},
334   → bg_scores :: AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat;
335   → remote=false)
336   new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources);
337   → iterate=1
338   new_sources=deepcopy(m.sources)
339
340   a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
341   → bg_scores, m.mix_matrix, true, true)
342
343   svec=[1:S ... ]
344
345   while new_log_Li ≤ contour && length(svec)>0 #until we produce a
346   → model more likely than the lh contour or no more sources to
347   → attempt infofill
348   new_sources=deepcopy(m.sources)
349   clean=Vector{Bool}(trues(0))
350   s = popat!(svec,rand(1:length(svec))) #randomly select a source
351   → to merge
352   pwm=new_sources[s][1]
353   fill_idx=findmin(get_pwm_info(pwm))[2]
354   fill_bases=[1,2,3,4]
355   fill_candidate=findmax(pwm[fill_idx,:])[2]
356   deleteat!(fill_bases, findfirst(b→b==fill_candidate,fill_bases))
357   new_sources[s][1][fill_idx,:]=-Inf;
358   → new_sources[s][1][fill_idx,fill_candidate]=0.
359   clean[m.mix_matrix[:,s]]=false
360   new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
361   → obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
362   → clean) #assess likelihood
363
364   while new_log_Li ≤ contour && length(fill_bases)>0
365     fill_candidate=popat!(fill_bases,rand(1:length(fill_bases)))
366     new_sources[s][1][fill_idx,:]=-Inf;
367     → new_sources[s][1][fill_idx,fill_candidate]=0.
368     new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
369     → obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
370     → clean) #assess likelihood
371   end
372 end

```

```

357
358     new_log_Li ≤ contour ? (blacklist=[info_fill]) :
→      (blacklist=Vector{Function}())
359
360     cons_check, cons_idxs = consolidate_check(new_sources)
361     cons_check ? (return ICA_PWM_Model("candidate","IF from
→      $(m.name)",new_sources, m.source_length_limits, m.mix_matrix,
→      new_log_Li, blacklist)) : (return consolidate_srcs(cons_idxs,
→      ICA_PWM_Model("candidate","IF from $(m.name)",new_sources,
→      m.source_length_limits, m.mix_matrix, new_log_Li, blacklist),
→      obs_array, obs_lengths, bg_scores, contour, models;
→      remote=remote))
362 end
363
364 full_perm_funcvec=[permute_source, permute_mix, perm_src_fit_mix,
→      fit_mix, random_decorrelate, shuffle_sources, accumulate_mix,
→      distance_merge, similarity_merge, reinit_src, erode_model, info_fill]

```

15.3.14 /src/permuation/permute_utilities.jl

```

1 ##BASIC UTILITY FUNCTIONS
2 #SOURCE PERMUTATION
3 function
4     permute_source_weights(source::Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}
5     ↪ shift_freq::AbstractFloat,
6     ↪ PWM_shift_dist::Distribution{Univariate,Continuous})
7     dirty=false; source_length=size(source[1],1)
8     new_source=deepcopy(source)
9
10    for source_pos in 1:source_length
11        if rand() ≤ shift_freq
12            pos_WM = exp.(source[1][source_pos,:]) #leaving logspace, get
13            ↪ the wm at that position
14            new_source[1][source_pos,:] = log.(wm_shift(pos_WM,
15            ↪ PWM_shift_dist)) #accumulate probabiltiy at a randomly
16            ↪ selected base, reassign in logspace and carry on
17            !dirty && (dirty=true)
18        end
19    end
20
21    if !dirty #if no positions were shifted, pick one and shift
22        rand_pos=rand(1:source_length)

```

```

17     pos_WM = exp.(source[1][rand_pos,:])
18     new_source[1][rand_pos,:]=log.(wm_shift(pos_WM, PWM_shift_dist))
19   end
20
21   return new_source
22 end
23
24   function
25     →  wm_shift(pos_WM::AbstractVector{<:AbstractFloat},
26     →  PWM_shift_dist::Distribution{Univariate,Continuous})
27     base_to_shift = rand(1:4) #pick a base to accumulate
28     →  probability
29     permute_sign = rand(-1:2:1)
30     shift_size = rand(PWM_shift_dist)
31     new_wm=zeros(4)
32
33     for base in 1:4 #ACGT
34       if base == base_to_shift
35         new_wm[base] =
36           clamp(0, #no lower than 0 prob
37             (pos_WM[base] #selected PWM posn
38             + permute_sign * shift_size), #randomly
39             →  permuted by size param
40             1) #no higher than prob 1
41       else
42         size_frac = shift_size / 3 #other bases
43         →  shifted in the opposite direction by 1/3
44         →  the shift accumulated at the base to
45         →  permute
46         new_wm[base] =
47           clamp(0,
48             (pos_WM[base]
49             - permute_sign * size_frac),
50             1)
51       end
52     end
53   end
54   new_wm = new_wm ./ sum(new_wm) #renormalise to sum 1
55   →  - necessary in case of clamping at 0 or 1
56   !isprobvec(new_wm) && throw(DomainError(new_wm, "Bad
57   →  weight vector generated in wm_shift!")) #throw
58   →  assertion exception if the position WM is invalid
59   return new_wm
60 end

```

```

50
51
52
53 function
  ↳ permute_source_length(source::Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer},
  ↳ prior::Union{<:AbstractVector{<:Dirichlet{<:AbstractFloat}},<:Bool},
  ↳ length_limits::UnitRange{<:Integer},
  ↳ permute_range::UnitRange{<:Integer}=LENGTHPERM_RANGE,
  ↳ uninformative::Dirichlet=Dirichlet([.25,.25,.25,.25]))
  source_PWM, prior_idx = source
  source_length = size(source_PWM,1)

54
55
56
57 permute_sign, permute_length = get_length_params(source_length,
  ↳ length_limits, permute_range)

58
59 permute_sign==1 ? permute_pos = rand(1:source_length+1) :
  permute_pos=rand(1:source_length-permute_length)

60
61
62 if permute_sign == 1 #if we're to add positions to the PWM
  ins_WM=zeros(permute_length,4)
  if prior==false
    for pos in 1:permute_length
      ins_WM[pos,:] = log.(transpose(rand(uninformative)))
    end
  else
    for pos in 1:permute_length
      prior_position=permute_pos+prior_idx-1
      prior_position<1 || prior_position>length(prior) ?
        ins_WM[pos,:] = log.(transpose(rand(uninformative)))
        :
      ins_WM[pos,:] =
        ↳ log.(transpose(rand(prior[prior_position])))
    end
  end
63
64 upstream_source=source_PWM[1:permute_pos-1,:]
65 downstream_source=source_PWM[permute_pos:end,:]
66 source_PWM=vcat(upstream_source,ins_WM,downstream_source)
67 permute_pos==1 && (prior_idx-=permute_length)
68 else #if we're to remove positions
  upstream_source=source_PWM[1:permute_pos-1,:]
  downstream_source=source_PWM[permute_pos+permute_length:end,:]
  source_PWM=vcat(upstream_source,downstream_source)
  permute_pos==1 && (prior_idx+=permute_length)
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84

```

```

85     end
86
87     return (source_PWM, prior_idx) #return a new source
88 end
89
90     function get_length_params(source_length::Integer,
91         → length_limits::UnitRange{<:Integer},
92         → permute_range::UnitRange{<:Integer})
93         extendable = length_limits[end]-source_length
94         contractable = source_length-length_limits[1]
95
96         if extendable == 0 && contractable > 0
97             permute_sign=-1
98         elseif contractable == 0 && extendable > 0
99             permute_sign=1
100        else
101            permute_sign = rand(-1:2:1)
102        end
103
104        permute_sign==1 && extendable<permute_range[end] &&
105            → (permute_range=permute_range[1]:extendable)
106        permute_sign== -1 && contractable<permute_range[end]
107            → && (permute_range=permute_range[1]:contractable)
108        permute_length = rand(permute_range)
109
110        return permute_sign, permute_length
111    end
112
113
114    function erode_source(source::Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer},length_li
115        pwm,prior_idx=source
116        infovec=get_pwm_info(pwm)
117        start_idx,end_idx=get_erosion_idxs(infovec, info_thresh,
118            → length_limits)
119
120        return new_source=(pwm[start_idx:end_idx,:], prior_idx+start_idx-1)
121    end
122
123    function get_pwm_info(pwm::AbstractMatrix{<:AbstractFloat};
124        → logsw::Bool=true)
125        wml=size(pwm,1)
126        infovec=zeros(wml)
127        for pos in 1:wml

```

```

121     logsw ? wvec=deepcopy(exp.(pwm[pos,:])) :
122         ↪ wvec=deepcopy(pwm[pos,:])
123     !isprobvec(wvec) && throw(DomainError(wvec, "Bad wvec in
124         ↪ get_pwm_info -Original sources must be in logspace!!"))
125     wvec.+=10^-99
126     infscore = (2.0 + sum([x*log(2,x) for x in wvec]))
127     infovec[pos]=infscore
128 end
129 return infovec
130
131 function get_erosion_idxs(infovec::AbstractVector{<:AbstractFloat},
132     ↪ info_thresh::AbstractFloat, length_limits::UnitRange{<:Integer})
133     srcl=length(infovec)
134     contractable = srcl-length_limits[1]
135     contractable ≤ 0 && throw(DomainError(contractable, "erode_source
136         ↪ passed a source at its lower length limit!"))
137     centeridx=findmax(infovec)[2]
138
139     start_idx=findprev(info→<(info,info_thresh),infovec,centeridx)
140     start_idx==nothing ? (start_idx=1) : (start_idx+=1)
141     end_idx=findnext(info→<(info, info_thresh),infovec,centeridx)
142     end_idx==nothing ? (end_idx=srcl) : (end_idx-=1)
143
144     pos_to_eroде=srcl-(end_idx-start_idx)
145     if pos_to_eroде > contractable
146         pos_to_restore = pos_to_eroде-contractable
147         while pos_to_restore>0
148             end_die=rand()
149             if end_die ≤ .5
150                 start_idx>1 && (pos_to_restore-=1; start_idx-=1)
151             else
152                 end_idx<srcl && (pos_to_restore-=1; end_idx+=1)
153             end
154         end
155     end
156
157     return start_idx, end_idx
158 end
159
160 #MIX MATRIX FUNCTIONS
161 function mixvec_decorrelate(mix::BitVector, moves::Integer)
162     new_mix=deepcopy(mix)

```

```

160     idxs_to_flip=rand(1:length(mix), moves)
161     new_mix[idxs_to_flip] *= .!mix[idxs_to_flip]
162     return new_mix
163 end
164
165 function mix_matrix_decorrelate(mix::BitMatrix, moves::Integer)
166     clean=Vector{Bool}(trues(size(mix,1)))
167     new_mix=deepcopy(mix)
168     indices_to_flip = rand(CartesianIndices(mix), moves)
169     new_mix[indices_to_flip] *= .!mix[indices_to_flip]
170     clean[unique([idx[1] for idx in indices_to_flip])] *= false #mark all
171     → obs that had flipped indices dirty
172     return new_mix, clean
173 end
174
175 # function most_dissimilar(mix1, mix2)
176 #     S1=size(mix1,2);S2=size(mix2,2)
177 #     dist_mat=zeros(S1,S2)
178 #     for s1 in 1:S1, s2 in 1:S2
179 #         dist_mat[s1,s2]=sum(mix1[:,s1].==mix2[:,s2])
180 #     end
181 #     scores=vec(sum(dist_mat,dims=1))
182 #     return findmin(scores)[2]
183 # end
184
185
186 function most_dissimilar(src_mixvec, target_mixmat)
187     src_sim = [sum(src_mixvec.==target_mixmat[:,s]) for s in
188     → 1:size(target_mixmat,2)] #compose array of elementwise equality
189     → comparisons between mixvectors and sum to score
190     merge_s=findmin(src_sim)[2] #source from merger model will be the one
191     → with the highest equality comparison score
192 end
193
194 function most_similar(src_mixvec, target_mixmat)
195     src_sim = [sum(src_mixvec.==target_mixmat[:,s]) for s in
196     → 1:size(target_mixmat,2)] #compose array of elementwise equality
197     → comparisons between mixvectors and sum to score
198     merge_s=findmax(src_sim)[2] #source from merger model will be the one
199     → with the highest equality comparison score
200 end

```

15.3.15 /src/utilities/model_display.jl

```

1 #THIS CODE BASED ON N. REDDY'S THICWEED.JL DISPLAY SCRIPT
2
3 #CONSTANTS
4 # each tuple specifies the x, y, fontsize, xscale to print the character
5 # in a 100×100 box at position 100,100.
6 charvals_dict = Dict{Char,Tuple}('A'⇒(88.5,199.,135.,1.09),
7                                'C'⇒(79.,196.,129.,1.19),
8                                'G'⇒(83.,196.,129.,1.14),
9                                'T'⇒(80.,199.,135.,1.24))
10
11 colour_dict = Dict{Char,String}('A'⇒"(0,128,0)", #green
12                                'C'⇒"(0,0,128)", #blue
13                                'G'⇒"(255,69,0)", #yellow-brown
14                                'T'⇒"(150,0,0)") #red
15
16 #char output params for string display of PWMs in nested sampler
   ↪ instrumentation
17 lwcs_vec=['a','c','g','t']
18 upcs_vec=['A','C','G','T']
19 cs_vec[:green,:blue,:yellow,:red]
20 thresh_dict=Dict([(0.,'_'),(.25,'.'),(.5,"lwcs"),(1,"upcs")])
21
22 source_left_x = 10
23 xlen = 20
24 yscale = 250
25 scale_factor = 0.9
26 ypixels_per_source = 250
27 ypad = 10
28 xpad = 20
29 xpixels_per_position = 40
30 fontsize1=60
31 fontsize2=40
32
33 #function to convert ICA_PWM_Model sources lists to PWM sequence logo
   ↪ diagrams
34 function
   ↪ logo_from_model(model::ICA_PWM_Model,svg_output::String;freq_sort::Bool=false)
35     source_tups = Vector{Tuple{AbstractFloat, Integer,
   ↪ Matrix{AbstractFloat}}}() #(%of sequences w/ source, prior index,
   ↪ weight matrix)
36     mix = model.mix_matrix #o x s

```

```

37   for (prior, source) in enumerate(model.sources)
38     push!(source_tups,
39       (sum(model.mix_matrix[:,prior])/size(model.mix_matrix)[1],
40        prior, exp.(source[1])))
41   end
42
43 freq_sort && sort(source_tups)
44
45 file = open(svg_output, "w")
46 write(file,
47   ← svg_header(xpad+xpixels_per_position*maximum([length(source[1])
48   ← for source in
49   ← model.sources]),ypixels_per_source*length(model.sources)+ypad))
50
51 curry = 0
52 for (frequency, index, source) in source_tups
53   curry += yscale
54   font1y = curry-190
55   ndig = ndigits(index+1)
56   write(file,
57     ← pwm_to_logo(source,source_left_x,curry,xlen,yscale*scale_factor))
58   write(file, "<text x=\"$10\" y=\"$font1y\""
59     ← font-family=\"Helvetica\" font-size=\"$fontsize1\""
60     ← font-weight=\"bold\" >$index</text>")
61   write(file, "<text x=\"$((20+$fontsize1*0.6*ndig))\""
62     ← y=\"$($font1y-$fontsize1+1.5*$fontsize2)\""
63     ← font-family=\"Helvetica\" font-size=\"$fontsize2\""
64     ← font-weight=\"bold\" >$frequency*100) % of
65     ← sequences</text>\n")
66 end
67
68 write(file, svg_footer())
69 @info "Logo written."
70 close(file)
71 end
72
73 function svg_header(canvas_size_x, canvas_size_y)
74   return """
75   <?xml version="1.0" standalone="no"?>
76   <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
77   "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
78   <svg width="$canvas_size_x" height="$canvas_size_y"
79     version="1.1" xmlns="http://www.w3.org/2000/svg">
```

```

68     """
69 end
70
71 function pwm_to_logo(source,xpos,ypos,xlen,yscale)
72     outstr = ""
73     for n in 1:size(source,1)
74         outstr *=
75             → print_weightvec_at_x_y(source[n,:],xpos+xlen*n,ypos,xlen,yscale)
76     end
77     outstr *= "\n"
78     return outstr
79 end
80
81 function pwmstr_to_io(io,source;log=true)
82     log && (source=exp.(source))
83     for pos in 1:size(source,1)
84         char,color=uni_wvec_params(source[pos,:])
85         printstyled(io, char; color=color)
86     end
87 end
88 function uni_wvec_params(wvec) #assign a single unicode char and color
89     ← symbol for the most informational position in a position weight
90     ← vector
91     wvec.+=10^-99
92     infoscore=(2.0 + sum([x*log(2,x) for x in wvec]))
93     infovec = [x*infoscore for x in wvec]
94     val,idx=findmax(infovec)
95     return char,color=get_char_by_thresh(idx,val)
96 end
97
98 function get_char_by_thresh(idx,val)
99     char = '?'; seen_thresh=0.
100    for (thresh,threshchar) in thresh_dict
101        val ≥ thresh && thresh ≥ seen_thresh &&
102            → (char=threshchar; seen_thresh=thresh)
103    end
104    char=='?' && println(val)
105    char=="lwcs" && (char=lwcs_vec[idx])
106    char=="upcs" && (char=upcs_vec[idx])
107    color=cs_vec[idx]
108
109    return char,color
110 end

```

```

107
108
109
110
111
112 function print_weightvec_at_x_y(wvec, xpos, ypos, xlen, yscale)
113     # xlen is the length occupied by that column
114     # yscale is the total height for 2 information bits i.e. the
115     # maximum height available for a "perfect" nucleotide
116     outstr = ""
117     basestr = "ACGT"
118     wvec.+=10^-99 #prevent log(0) = -Inf
119     wvec = [x/sum(wvec) for x in wvec] #renorm
120     infscore = (2.0 + sum([x*log(2,x) for x in wvec]))
121     if infscore==0.0
122         return ""
123     end
124     wvec = [x*infscore*yscale/2.0 for x in wvec]
125     # at this point, the sum of all wvec is a maximum of yscale
126     wveclist = [(wvec[n],basestr[n]) for n in 1:4]
127     wveclist = sort(wveclist)
128     curr_ypos = ypos
129     for n in 1:4
130         curr_ypos -= wveclist[n][1]
131         outstr *=
132             → print_char_in_rect(wveclist[n][2],xpos,curr_ypos,xlen,wveclist[n][1])
133     end
134     return outstr
135 end
136 blo = 1
137
138 function print_char_in_rect(c,x,y,width,height)
139     raw_x, raw_y, raw_fontsize, raw_xsclae = charvals_dict[c]
140     raw_x = (raw_x*raw_xsclae-100)/raw_xsclae
141     raw_y = raw_y-100
142
143     xscale = width/100.0 * raw_xsclae
144     yscale = height/100.0
145
146     scaled_x = x/xscale + raw_x
147     scaled_y = y/yscale + raw_y
148

```

```

149     return "<text x=\"$scaled_x\" y=\"$scaled_y\""
150     ↵   font-size=\"$raw_fontsize\" font-weight=\"$bold\"
151     ↵   font-family=\"$Helvetica\" fill=\"$rgb\"*colour_dict[c]*\""
152     ↵   transform=\"scale($xscale,$yscale)\">$c</text>\n"
153   end
154
155
156 function svg_footer()
157   return "</svg>"
158 end

```

15.3.16 /src/utilities/ns_progressmeter.jl

```

1 #UTILITY REPORTS WORKER NUMBER AND CURRENT ITERATE
2 mutable struct ProgressNS{T<:Real} <: AbstractProgress
3     interval::T
4     dt::AbstractFloat
5     start_it::Integer
6     counter::Integer
7     triggered::Bool
8     tfirst::AbstractFloat
9     tlast::AbstractFloat
10    tstop::AbstractFloat
11    printed::Bool      # true if we have issued at least one status
12    ↵      update
13    desc::AbstractString # prefix to the percentage, e.g. "Computing ... "
14    color::Symbol        # default to green
15    output::IO           # output stream into which the progress is
16    ↵      written
17    numprintedvalues::Integer # num values printed below progress in
18    ↵      last iteration
19    offset::Integer       # position offset of progress bar
20    ↵      (default is 0)
21
22    e::IPM_Ensemble
23    wm::Worker_Monitor
24    tuner::Permute_Tuner
25    top_m::ICA_PWM_Model
26
27    mean_stp_time::AbstractFloat
28
29    wk_disp::Bool
30    tuning_disp::Bool

```



```

68     printed,
69     desc,
70     color,
71     output,
72     0,
73     offset,
74     e,
75     wm,
76     tuner,
77     top_m,
78     0.,
79     wk_disp,
80     tuning_disp,
81     conv_plot,
82     ens_disp,
83     lh_disp,
84     liwi_disp,
85     src_disp,
86     nsrcts,
87     disp_rotate_inst,
88     zeros(CONVERGENCE_MEMORY))
89   end
90 end
91
92 function ProgressNS(e::IPM_Ensemble, wm::Worker_Monitor,
93   → tuner::Permute_Tuner, interval::Real; dt::Real=0.1,
94   → desc::AbstractString="Nested Sampling:: ", color::Symbol=:green,
95   → output::IO=stderr, offset::Integer=0, start_it::Integer=1,
96   → wk_disp::Bool=false, tuning_disp::Bool=false, conv_plot::Bool=false,
97   → lh_disp::Bool=false, liwi_disp::Bool=false, ens_disp::Bool=false,
98   → src_disp::Bool=false, nsrcts=0,
99   → disp_rotate_inst::Vector{Any}=[false,0,0,Vector{Vector{Symbol}}()])
100   top_m = deserialize(e.models[findmax([model.log_Li for model in
101     → e.models])[2]].path)
102
103   return ProgressNS{typeof(interval)}(e, top_m, wm, tuner, interval,
104     → dt=dt, desc=desc, color=color, output=output, offset=offset,
105     → start_it=start_it, wk_disp=wk_disp, tuning_disp=tuning_disp,
106     → conv_plot=conv_plot, lh_disp=lh_disp, liwi_disp=liwi_disp,
107     → ens_disp=ens_disp, src_disp=src_disp, nsrcts=nsrcts,
108     → disp_rotate_inst=disp_rotate_inst)
109 end
110
111

```

```

98
99 function update!(p::ProgressNS, val, thresh; options ... )
100     p.counter += 1
101
102     p.tstp=time()-p.tlast
103     popfirst!(p.tuner.time_history)
104     push!(p.tuner.time_history, p.tstp)
105
106     p.interval = val - thresh
107     popfirst!(p.convergence_history)
108     push!(p.convergence_history, p.interval)
109
110     p.top_m.name ≠ basename(p.e.models[findmax([model.log_Li for model
111         in p.e.models])[2]].path) &&
112         (p.top_m=deserialize(p.e.models[findmax([model.log_Li for model
113             in p.e.models])[2]].path))
114
114
115 function updateProgress!(p::ProgressNS; offset::Integer = p.offset, keep
116     = (offset == 0))
117     p.offset = offset
118     t = time()
119     p.disp_rotate_inst[1] && p.counter%p.disp_rotate_inst[2] == 0 &&
120         rotate_displays(p)
121
121
122     if p.interval ≤ 0 && !p.triggered
123         p.triggered = true
124         if p.printed
125             p.triggered = true
126             dur = durationstring(t-p.tfirst)
127             msg = @sprintf "%s Converged. Time: %s (%d iterations). logZ:
128                         %s\n" p.desc dur p.counter p.e.log_Zi[end]
129
130             print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
131             move_cursor_up_while_clearing_lines(p.output,
132                 p.numprintedvalues)
133             upper_lines=display_upper_dash(p)
134             printover(p.output, msg, :magenta)
135             lower_lines=display_lower_dash(p)
136
137             p.numprintedvalues=upper_lines + lower_lines + 1

```

```

134
135     if keep
136         println(p.output)
137     else
138         print(p.output, "\r\u1b[A" ^ (p.offset +
139             → p.numprintedvalues))
140     end
141   return
142 end
143
144 if t > p.tlast+p.dt && !p.triggered
145   p.counter < CONVERGENCE_MEMORY ?
146     → mean_step_time=mean(p.tuner.time_history[end-(p.counter-1):end])
147     → :
148     → mean_step_time=mean(p.tuner.time_hist
149   msg = @sprintf "%s Iterate: %s Recent step time μ: %s Convergence
150   → Interval: %g\n" p.desc p.counter hmss(mean_step_time)
151   → p.interval
152
153   print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
154   move_cursor_up_while_clearing_lines(p.output, p.numprintedvalues)
155   upper_lines=display_upper_dash(p)
156   printover(p.output, msg, p.color)
157   lower_lines=display_lower_dash(p)
158
159   p.numprintedvalues=upper_lines + lower_lines + 1
160   print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))
161
162   # Compensate for any overhead of printing. This can be
163   # especially important if you're running over a slow network
164   # connection.
165   p.tlast = t + 2*(time()-t)
166   p.printed = true
167 end
168 end
169
170 function rotate_displays(p)
171   curr_inst=p.disp_rotate_inst[3]
172   curr_inst+1>length(p.disp_rotate_inst[4]) ?
173     → next_inst=1 : next_inst=curr_inst+1
174   next_disps=p.disp_rotate_inst[4][next_inst]

```

```

169         for disp in
170             → [:wk_disp,:tuning_disp,:conv_plot,:ens_disp,:lh_disp,:liwi_di
171             disp in next_disps ? setproperty!(p, disp, true)
172             → : setproperty!(p, disp, false)
173         end
174         p.disp_rotate_inst[3]=next_inst
175     end
176
177     function hmss(dt)
178         dt<0 ? (dt=-dt; prfx="-") : (prfx="")
179         isnan(dt) && return "NaN"
180         (h,r) = divrem(dt,60*60)
181         (m,r) = divrem(r, 60)
182         (isnan(h)||isnan(m)||isnan(r)) && return "NaN"
183         string(prfx,Int(h),":",Int(m),":",Int(ceil(r)))
184     end
185
186     function display_upper_dash(p::ProgressNS)
187         wklines = tunelines = cilines = 0
188         p.wk_disp && (wklines=show(p.output, p.wm,
189             → progress=true);println())
190         p.tuning_disp && (tunelines=show(p.output, p.tuner,
191             → progress=true);println())
192         if p.conv_plot
193             → ciplot=lineplot([p.counter-(length(p.convergence_history)
194             → p.convergence_history, title="Convergence
195             → Interval Recent History",
196             → xlabel="Iterate",ylabel="CI", color=:yellow)
197             cilines=nrows(ciplot.graphics)+5
198             show(p.output, ciplot); println()
199         end
200         return wklines + tunelines + cilines
201     end
202
203     function display_lower_dash(p::ProgressNS)
204         lhlines = liwilines = ensemblelines = srclines = 0
205         if p.lh_disp
206             lhplot=lineplot(p.e.log_Li[2:end], title="Contour
207             → History", xlabel="Iterate", color=:magenta,
208             → name="Ensemble logLH")

```

```

200             lineplot!(lhplot, [p.e.naive_lh for it in
201             ↪ 1:length(p.e.log_Li[2:end])], name="Naive
202             ↪ logLH")
203             lhlines=nrows(lhplot.graphics)+5
204             show(p.output, lhplot); println()
205         end
206         if p.liwi_disp && p.counter>2
207
208             ↪ liwiplot=lineplot([max(2,p.counter-(CONVERGENCE_MEMORY-1)
209             ↪ title="Recent iterate evidentiary weight",
210             ↪ xlabel="Iterate", name="Ensemble log Liwi",
211             ↪ color=:cyan)
212             liwilines=nrows(liwiplot.graphics)+5
213             show(p.output, liwiplot); println()
214         end
215         p.ens_disp && (ensemblelines=show(p.output, p.e,
216             ↪ progress=true))
217         p.src_disp && (println("MLE Model
218             ↪ Sources:");srclines=show(p.output, p.top_m,
219             ↪ nsr=p.no_displayed_srcs, progress=true))
220     return lhlines + liwilines + ensemblelines + srclines
221   end

```

15.3.17 /src/utilities/worker_diagnostics.jl

```

1 struct Worker_Monitor
2     idx::Dict{Integer, Integer}
3     persist::BitMatrix
4     last_seen::Matrix{Float64}
5 end
6
7 function Worker_Monitor(wk_pool::Vector{<:Integer})
8     idx=Dict{Integer, Integer}()
9     for (n,wk) in enumerate(wk_pool)
10        idx[wk]=n
11    end
12    persist=trueones(1,length(wk_pool))
13    last_seen=[time() for x in 1:1, y in 1:length(wk_pool)]
14    return Worker_Monitor(idx, persist, last_seen)
15 end
16
17 function update_worker_monitor!(mon, wk, persist)

```

```

18     mon.persist[1,mon.idx[wk]]=persist
19     mon.last_seen[1,mon.idx[wk]]=time()
20 end
21
22 function Base.show(io::IO, mon::Worker_Monitor; progress=false)
23     printstyled("Worker Diagnostics\n", bold=true)
24     pers=heatmap(float.(mon.persist), colormap=persistcolor,
25         → title="Persistence", labels=false)
26     ls=heatmap([time()-ls for ls in mon.last_seen], title="Last
27         → Seen", labels=false)
28     show(pers)
29     println()
30     show(ls)
31     progress && return nrows(pers.graphics)+nrows(ls.graphics)+7
32 end
33
34         function persistcolor(z, zmin, zmax)
35             z==1. && return 154
36             z==0. && return 160
37         end

```

15.3.18 /test/runtests.jl

```

1 @info "Loading test packages ... "
2
3 using BioMotifInference, BioBackgroundModels, BioSequences,
4     ↳ Distributions, Distributed, Random, Serialization, Test
5 import StatsFuns: logsumexp, logaddexp
6 import BioMotifInference:estimate_dirichlet_prior_on_wm,
7     ↳ assemble_source_priors, init_logPWM_sources, wm_shift,
8     ↳ permute_source_weights, get_length_params, permute_source_length,
9     ↳ get_pwm_info, get_erosion_idxs, erode_source, init_mix_matrix,
10    ↳ mixvec_decorrelate, mix_matrix_decorrelate, most_dissimilar,
11    ↳ most_similar, revcomp_pwm, score_sources_ds!, score_sources_ss!,
12    ↳ weave_scores_ss!, weave_scores_ds!, IPM_likelihood,
13    ↳ consolidate_check, consolidate_srcs, pwm_distance, permute_source,
14    ↳ permute_mix, perm_src_fit_mix, fit_mix, random_decorrelate,
15    ↳ reinit_src, erode_model, reinit_src, shuffle_sources, accumulate_mix,
16    ↳ distance_merge, similarity_merge, info_fill, converge_ensemble!,
17    ↳ reset_ensemble!, Permute_Tuner, PRIOR_WT, TUNING_MEMORY,
18    ↳ CONVERGENCE_MEMORY, tune_weights!, update_weights!, clamp_pvec!
19 import Distances: euclidean

```

```
7
8 @info "Beginning tests ... "
9 using Random
10 Random.seed!(786)
11 O=1000;S=50
12
13 include("pwm_unit_tests.jl")
14 include("mix_matrix_unit_tests.jl")
15 include("likelihood_unit_tests.jl")
16 include("consolidate_unit_tests.jl")
17 include("permute_func_tests.jl")
18 include("permute_tuner_tests.jl")
19 include("ensemble_tests.jl")
20 @info "Tests complete!"
```

15.3.19 /test/spike_recovery.jl

```
1 #Testbed for synthetic spike recovery from example background
2
3 using nnlearn, BGHMM, HMMBase, Distributions, Random, Serialization,
4   ↳ Distributions
5
6 Random.seed!(786)
7
8 #CONSTANTS
9 no_obs=1000
10 obsl=140
```

```

11 hmm=HMM{Univariate,Float64}([0.4016518533961019, 0.2724399569450827,
→ 0.3138638675018568, 0.012044322156962559], [3.016523036789942e-9
→ 2.860631288328858e-6 0.2299906524188302 0.7700064839333549;
→ 3.0102278323431375e-11 0.7494895424906354 0.23378615437778671
→ 0.016724303101477486; 8.665894321573098e-17 0.2789950381410553
→ 0.7141355461949104 0.006869415664033568; 0.006872526597796038
→ 0.016052425322133648 0.017255179541768192 0.9598198685383041],
→ [Categorical([0.1582723599684065, 0.031949729618356536,
→ 0.653113286526948, 0.15666462388628763]),
→ Categorical([0.4610499748798372, 0.2613013005680122,
→ 0.15161801872560146, 0.12603070582654768]),
→ Categorical([0.08601960130086236, 0.13869192872427524,
→ 0.26945182329686523, 0.5058366466779973]),
→ Categorical([0.3787366527613563, 0.11034604356978756,
→ 0.1119586976058099, 0.3989586060630392])))

12
13 struc_sig=[.1 .7 .1 .1
14           .1 .1 .1 .7
15           .1 .7 .1 .1]
16 periodicity=8
17 struc_frac_obs=.35
18
19 tata_box=[.05 .05 .05 .85
20           .85 .05 .05 .05
21           .05 .05 .05 .85
22           .85 .05 .05 .05
23           .425 .075 .075 .425
24           .85 .05 .05 .05
25           .425 .075 .075 .425]
26 tata_frac_obs=.7
27 tata_recur_range=1:4
28
29 combined_ensemble = "/bench/PhD/NGS_binaries/nnlearn/combined_ensemble"
30
31 #JOB CONSTANTS
32 const position_size = 141
33 const ensemble_size = 100
34 const no_sources = 3
35 const source_min_bases = 3
36 const source_max_bases = 12
37 @assert source_min_bases < source_max_bases
38 const source_length_range= source_min_bases:source_max_bases
39 const mixing_prior = .3

```

```

40 @assert mixing_prior ≥ 0 & mixing_prior ≤ 1
41 const models_to_permute = ensemble_size * 3
42
43 job_sets=[[
44 ([[
45     ("PS", (no_sources)),
46     ("PM", (no_sources)),
47     ("PSFM", (no_sources)),
48     ("PSFM", (no_sources*10, .8, 1.)),
49     ("FM", (())),
50     ("DM", (no_sources)),
51     ("SM", (no_sources)),
52     ("RD", (no_sources)),
53     ("RI", (no_sources)),
54     ("EM", (no_sources))
55 ], [.025, .025, .025, .025, .775, .025, .025, .025, .025, .025]),,
56 ([[
57     ("PS", (no_sources)),
58     ("PM", (no_sources)),
59     ("PSFM", (no_sources)),
60     ("PSFM", (no_sources, 0., 1.)),
61     ("PSFM", (no_sources, 0.8, .0)),
62     ("FM", (())),
63     ("DM", (no_sources)),
64     ("SM", (no_sources)),
65     ("RD", (no_sources)),
66     ("RI", (no_sources*10)),
67     ("EM", (no_sources))
68 ], [.05, .15, .15, .10, .05, .20, 0.0, 0.05, 0.15, 0.10, .0]),,
69 ]
70 job_limit=4
71
72 const prior_wt=1.1
73
74 #FUNCTIONS
75 function setup_obs(hmm, no_obs, obsl)
76     obs=vec(Int64.(rand(hmm,obsl)[2]))
77     for o in 2:no_obs
78         obs=hcat(obs, vec(Int64.(rand(hmm,obsl)[2])))
79     end
80     obs=vcat(obs,zeros(Int64,1,no_obs))
81     return obs
82 end

```

```

83
84 function spike_irreg!(obs, source, frac_obs, recur)
85     truth=Vector{Int64}()
86     for o in 1:size(obs,2)
87         if rand()<frac_obs
88             push!(truth, o)
89             for r in rand(recur)
90                 rand()<.5 && (source=nnlearn.revcomp_pwm(source))
91                 pos=rand(1:size(obs,1)-1)
92                 pwm_ctr=1
93                 while pos ≤ size(obs,1)-1&& pwm_ctr ≤ size(source,1)
94                     obs[pos,o]=rand(Categorical(source[pwm_ctr,:]))
95                     pos+=1
96                     pwm_ctr+=1
97                 end
98             end
99         end
100    return truth
101 end
102
103
104 function spike_struc!(obs, source, frac_obs, periodicity)
105     truth=Vector{Int64}()
106     truthpos=Vector{Vector{Int64}}()
107     for o in 1:size(obs,2)
108         if rand()<frac_obs
109             push!(truth, o)
110             rand()<.5 && (source=nnlearn.revcomp_pwm(source))
111             pos=rand(1:periodicity)
112             posvec=Vector{Int64}()
113             while pos ≤ size(obs,1)
114                 pos_ctrl=pos
115                 pwm_ctrl=1
116                 while pos_ctrl ≤ size(obs,1)-1&&pwm_ctrl ≤ size(source,1)
117                     push!(posvec,pos_ctrl)
118                     base=rand(Categorical(source[pwm_ctrl,:]))
119                     obs[pos_ctrl,o]=base
120                     pos_ctrl+=1
121                     pwm_ctrl+=1
122                 end
123                 pos+=periodicity+pwm_ctrl
124             end
125             push!(truthpos,posvec)

```

```

126     end
127   end
128   return truth,truthpos
129 end
130
131 function get_BGHMM_lhs(obs,hmm)
132   lh_mat=zeros(size(obs,1)-1,size(obs,2))
133   for o in 1:size(obs,2)
134     obso=zeros(Int64,size(obs,1),1)
135     obso[1:end,1] = obs[:,o]
136
137     → lh_mat[:,o]=BGHMM.get_BGHMM_symbol_lh(Matrix(transpose(obso)),hmm)
138   end
139   return lh_mat
140 end
141 @info "Setting up synthetic observation set ... "
142 obs=setup_obs(hmm, no_obs, obsl)
143 struc_truth,struc_postruth=spike_struc!(obs, struc_sig, struc_frac_obs,
144   → periodicity)
145 irreg_truth=spike_irreg!(obs, tata_box, tata_frac_obs, tata_recur_range)
146
147 @info "Calculating background likelihood matrix ... "
148 bg_lhs=get_BGHMM_lhs(obs,hmm)
149
150 @info "Assembling source priors ... "
151 #prior_array= [struc_sig, tata_box]
152 prior_array= Vector{Matrix{Float64}}(){}
153 source_priors = nnlearn.assemble_source_priors(no_sources, prior_array,
154   → prior_wt, source_length_range)
155
156 @info "Assembling ensemble ... "
157 path=randstring()
158 if isfile(string(path,'/',"ens"))
159   ens = deserialize(string(path,'/',"ens"))
160   job_set_thresh=[-Inf,ens.naive_lh]
161   param_set=(job_sets,job_set_thresh,job_limit)
162
163 @info "Converging ensemble ... "

```

```
164 nnlearn.ns_converge!(ens, param_set, models_to_permute, .0001,
    ↪ model_display=no_sources, backup=(true,5))
165
166 rm(path,recursive=true)
167
168 #811973
```

Bibliography

- [ABS⁺10] W. Ted Allison, Linda K. Barthel, Kristina M. Skebo, Masaki Takechi, Shoji Kawamura, and Pamela A. Raymond. Ontogeny of cone photoreceptor mosaics in zebrafish. *The Journal of Comparative Neurology*, 518(20):4182–4195, October 2010.
- [AH09] Michalis Agathocleous and William A. Harris. From progenitors to differentiated cells in the vertebrate retina. *Annual Review of Cell and Developmental Biology*, 25:45–69, 2009.
- [AHW⁺20] Afnan Azizi, Anne Herrmann, Yinan Wan, Salvador JRP Buse, Philipp J Keller, Raymond E Goldstein, and William A Harris. Nuclear crowding and nonlinear diffusion during interkinetic nuclear migration in the zebrafish retina. 9(58635):31, 2020.
- [ALHP07] Michalis Agathocleous, Morgane Locker, William A. Harris, and Muriel Perron. A General Role of Hedgehog in the Regulation of Proliferation. *Cell Cycle*, 6(2):156–159, January 2007.
- [AR08] Ruben Adler and Pamela A. Raymond. Have we achieved a unified model of photoreceptor cell fate specification in vertebrates? *Brain Research*, 1192:134–150, February 2008.
- [BA02] Kenneth P. Burnham and David Raymond Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer, New York, 2nd ed edition, 2002.
- [BB20] M.J. Brewer and A. Butler. Model selection and the cult of AIC. In *Departmental Seminar*, University of Wollongong, Australia, February 2020.
- [BGL⁺10] Klaus A. Becker, Prachi N. Ghule, Jane B. Lian, Janet L. Stein, Andre J. van Wijnen, and Gary S. Stein. Cyclin D2 and the CDK substrate p220^{NPAT} are required for self-renewal of human embryonic stem cells. *Journal of Cellular Physiology*, 222(2):456–464, February 2010.
- [Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 2006.
- [BJ79] D. H. Beach and Marcus Jacobson. Influences of thyroxine on cell proliferation in the retina of the clawed frog at different ages. *The Journal of Comparative Neurology*, 183(3):615–623, February 1979.
- [BNL⁺96] M. Burmeister, J. Novak, M. Y. Liang, S. Basu, L. Ploder, N. L. Hawes, D. Vidgen, F. Hoover, D. Goldman, V. I. Kalnins, T. H. Roderick, B. A. Taylor, M. H. Hankin,

- and R. R. McInnes. Ocular retardation mouse caused by Chx10 homeobox null allele: Impaired retinal progenitor proliferation and bipolar cell differentiation. *Nature Genetics*, 12(4):376–384, April 1996.
- [BRD⁺15] Henrik Boije, Steffen Rulands, Stefanie Dudczig, Benjamin D. Simons, and William A. Harris. The Independent Probabilistic Firing of Transcription Factors: A Paradigm for Clonal Variability in the Zebrafish Retina. *Developmental Cell*, 34(5):532–543, September 2015.
- [Bri10] Ingo Brigandt. Beyond reduction and pluralism: Toward an epistemology of explanatory integration in biology. *Erkenntnis*, 73(3):295–311, 2010.
- [CAH⁺14] L. Centanin, J.-J. Ander, B. Hoeckendorf, K. Lust, T. Kellner, I. Kraemer, C. Urbany, E. Hasel, W. A. Harris, B. D. Simons, and J. Wittbrodt. Exclusive multipotency and preferential asymmetric divisions in post-embryonic neural stem cells of the fish retina. *Development*, 141(18):3472–3482, September 2014.
- [CAI⁺04] Brenda LK Coles, Brigitte Angénieux, Tomoyuki Inoue, Katia Del Rio-Tsonis, Jason R. Spence, Roderick R. McInnes, Yvan Arsenijevic, and Derek van der Kooy. Facile isolation and the characterization of human retinal stem cells. *Proceedings of the National Academy of Sciences of the United States of America*, 101(44):15772–15777, 2004.
- [CAR⁺15] Renee W. Chow, Alexandra D. Almeida, Owen Randallett, Caren Norden, and William A. Harris. Inhibitory neuron migration and IPL formation in the developing zebrafish retina. *Development*, 142(15):2665–2677, August 2015.
- [Cav18] Florencia Cavodeassi. Dynamic Tissue Rearrangements during Vertebrate Eye Morphogenesis: Insights from Fish Models. *Journal of Developmental Biology*, 6(1):4, February 2018.
- [CAY⁺96] Constance L. Cepko, Christopher P. Austin, Xianjie Yang, Macrene Alexiades, and Diala Ezzeddine. Cell fate determination in the vertebrate retina. *Proceedings of the National Academy of Sciences*, 93(2):589–595, 1996.
- [CBR03] Michel Cayouette, Ben A. Barres, and Martin Raff. Importance of intrinsic mechanisms in cell fate decisions in the developing rat retina. *Neuron*, 40(5):897–904, December 2003.
- [CC07] Bo Chen and Constance L Cepko. Requirement of histone deacetylase activity for the expression of critical photoreceptor genes. *BMC Developmental Biology*, 7(1):78, 2007.
- [CCP09] J. M. Catchen, J. S. Conery, and J. H. Postlethwait. Automated identification of conserved synteny after whole-genome duplication. *Genome Research*, 19(8):1497–1505, August 2009.
- [CCY⁺05] Florencia Cavodeassi, Filipa Carreira-Barbosa, Rodrigo M. Young, Miguel L. Concha, Miguel L. Allende, Corinne Houart, Masazumi Tada, and Stephen W. Wilson. Early Stages of Zebrafish Eye Formation Require the Coordinated Activity of Wnt11, Fz5, and the Wnt/β-Catenin Pathway. *Neuron*, 47(1):43–56, July 2005.

- [CHB⁺08] Hannah H. Chang, Martin Hemberg, Mauricio Barahona, Donald E. Ingber, and Sui Huang. Transcriptome-wide noise controls lineage choice in mammalian progenitor cells. *Nature*, 453(7194):544–547, May 2008.
- [CHW11] Lázaro Centanin, Burkhard Hoeckendorf, and Joachim Wittbrodt. Fate Restriction and Multipotency in Retinal Stem Cells. *Cell Stem Cell*, 9(6):553–562, December 2011.
- [CYV⁺08] Anna M. Clark, Sanghee Yun, Eric S. Veien, Yuan Y. Wu, Robert L. Chow, Richard I. Dorsky, and Edward M. Levine. Negative regulation of Vsx1 by its paralog Chx10/Vsx2 is conserved in the vertebrate retina. *Brain Research*, 1192:99–113, February 2008.
- [Dar88] Charles Darwin. *The Origin of Species by Means of Natural Selection, or, The Preservation of Favoured Races in the Struggle for Life*. J. Murray,, London :, 1888.
- [DC00] Michael A. Dyer and Constance L. Cepko. Control of Müller glial cell proliferation and activation following retinal injury. *Nature Neuroscience*, 3(9):873–880, September 2000.
- [DCRH97] R. I. Dorsky, W. S. Chang, D. H. Rapaport, and W. A. Harris. Regulation of neuronal diversity in the Xenopus retina by Delta signalling. *Nature*, 385(6611):67–70, January 1997.
- [DFWV⁺97] MARCO Di Frusco, Hans Weiher, Barbara C. Vanderhyden, Takashi Imai, Tadahiro Shioomi, T. A. Hori, Rudolf Jaenisch, and Douglas A. Gray. Proviral inactivation of the Npat gene of Mpv 20 mice results in early embryonic arrest. *Molecular and cellular biology*, 17(7):4080–4086, 1997.
- [DH05] T. A. Down and Tim J.P. Hubbard. NestedMICA: Sensitive inference of over-represented motifs in nucleic acid sequence. *Nucleic Acids Research*, 33(5):1445–1453, March 2005.
- [DHM07] Persi Diaconis, Susan Holmes, and Richard Montgomery. Dynamical Bias in the Coin Toss. *SIAM Review*, 49(2):211–235, January 2007.
- [DPCH03] Tilak Das, Bernhard Payer, Michel Cayouette, and William A. Harris. In vivo time-lapse imaging of cell divisions during neurogenesis in the developing zebrafish retina. *Neuron*, 37(4):597–609, 2003.
- [DRH95] Richard I Dorsky, David H Rapaport, and William A Harris. Xotch inhibits cell differentiation in the xenopus retina. *Neuron*, 14(3):487–496, March 1995.
- [Eag18] Antony Eagle. Chance versus Randomness. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2018 edition, 2018.
- [EHML09] Ted Erclik, Volker Hartenstein, Roderick R. McInnes, and Howard D. Lipshitz. Eye evolution at high resolution: The neuron as a unit of homology. *Developmental Biology*, 332(1):70–79, August 2009.
- [ESY⁺17] Peter Engerer, Sachihiro C Suzuki, Takeshi Yoshimatsu, Prisca Chapouton, Nancy Obeng, Benjamin Odermatt, Philip R Williams, Thomas Misgeld, and Leanne Godinho. Uncoupling of neurogenesis and differentiation during retinal development. *The EMBO Journal*, 36(9):1134–1146, May 2017.

- [Fag13] Melinda Bonnie Fagan. *Philosophy of Stem Cell Biology: Knowledge in Flesh and Blood.* New Directions in the Philosophy of Science. Palgrave Macmillan, Hounds mills, Basingstoke, Hampshire, 2013.
- [Fag15] Melinda Bonnie Fagan. Collaborative explanation and biological mechanisms. *Studies in History and Philosophy of Science Part A*, 52:67–78, August 2015.
- [FEF⁺09] Curtis R. French, Timothy Erickson, Danielle V. French, David B. Pilgrim, and Andrew J. Waskiewicz. Gdf6a is required for the initiation of dorsal–ventral retinal patterning and lens development. *Developmental Biology*, 333(1):37–47, September 2009.
- [Fey93] Paul Feyerabend. *Against Method.* Verso, London ; New York, 3rd ed edition, 1993.
- [FH08] Farhan Feroz and M. P. Hobson. Multimodal nested sampling: An efficient and robust alternative to MCMC methods for astronomical data analysis. *Monthly Notices of the Royal Astronomical Society*, 384(2):449–463, January 2008.
- [FHB09] F. Feroz, M. P. Hobson, and M. Bridges. MultiNest: An efficient and robust Bayesian inference tool for cosmology and particle physics. *Monthly Notices of the Royal Astronomical Society*, 398(4):1601–1614, October 2009.
- [FK12] Chikara Furusawa and Kunihiko Kaneko. A Dynamical-Systems View of Stem Cell Biology. *Science*, 338(6104):215–217, October 2012.
- [FR00] Andy J. Fischer and Thomas A. Reh. Identification of a Proliferating Marginal Zone of Retinal Progenitors in Postnatal Chickens. *Developmental Biology*, 220(2):197–210, April 2000.
- [FR03] Andy J. Fischer and Thomas A. Reh. Potential of Müller glia to become neurogenic retinal progenitor cells: Retinal Müller Glia as a Source of Stem Cells. *Glia*, 43(1):70–76, July 2003.
- [GBB⁺03] G. Gao, A. P. Bracken, K. Burkard, D. Pasini, M. Classon, C. Attwooll, M. Sagara, T. Imai, K. Helin, and J. Zhao. NPAT Expression Is Regulated by E2F and Is Essential for Cell Cycle Progression. *Molecular and Cellular Biology*, 23(8):2821–2833, April 2003.
- [GDL⁺09] Prachi N. Ghule, Zbigniew Dominski, Jane B. Lian, Janet L. Stein, Andre J. van Wijnen, and Gary S. Stein. The subnuclear organization of histone gene regulatory proteins and 3' end processing factors of normal somatic and embryonic stem cells is compromised in selected human cancer cell types. *Journal of Cellular Physiology*, 220(1):129–135, July 2009.
- [Geh96] Walter J. Gehring. The master control gene for morphogenesis and evolution of the eye. *Genes to Cells*, 1(1):11–15, January 1996.
- [GJH⁺17] Mahdi Golkaram, Jiwon Jang, Stefan Hellander, Kenneth S. Kosik, and Linda R. Petzold. The Role of Chromatin Density in Cell Population Heterogeneity during Stem Cell Differentiation. *Scientific Reports*, 7(1):13307, October 2017.

- [GNB08] Nathan J. Gosse, Linda M. Nevin, and Herwig Baier. Retinotopic order in the absence of axon competition. *Nature*, 452(7189):892–895, April 2008.
- [Gol17] Sheldon Goldstein. Bohmian Mechanics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2017 edition, 2017.
- [GS07] Frédéric Gaillard and Yves Sauvé. Cell-based therapy for retina degeneration: The promise of a cure. *Vision Research*, 47(22):2815–2824, October 2007.
- [GTR⁺10] D. M. Gilbert, S.-I. Takebayashi, T. Ryba, J. Lu, B. D. Pope, K. A. Wilson, and I. Hiratani. Space and Time in the Nucleus: Developmental Control of Replication Timing and Chromosome Architecture. *Cold Spring Harbor Symposia on Quantitative Biology*, 75(0):143–153, January 2010.
- [GWC⁺07] Leanne Godinho, Philip R. Williams, Yvonne Claassen, Elayne Provost, Steven D. Leach, Maarten Kamermans, and Rachel O.L. Wong. Nonapical Symmetric Divisions Underlie Horizontal Cell Layer Formation in the Developing Retina In Vivo. *Neuron*, 56(4):597–603, November 2007.
- [GZC⁺11] Francisco L. A. F. Gomes, Gen Zhang, Felix Carbonell, José A. Correa, William A. Harris, Benjamin D. Simons, and Michel Cayouette. Reconstruction of rat retinal progenitor cell lineages in vitro reveals a surprising degree of stochasticity in cell fate decisions. *Development (Cambridge, England)*, 138(2):227–235, January 2011.
- [HBEH88] Christine E. Holt, Thomas W. Bertsch, Hillary M. Ellis, and William A. Harris. Cellular Determination in the Xenopus Retina is Independent of Lineage and Birth Date. *Neuron*, 1(1):15–26, March 1988.
- [HCG95] G Halder, P Callaerts, and W. Gehring. Induction of ectopic eyes by targeted expression of the eyeless gene in Drosophila. *Science*, 267(5205):1788–1792, March 1995.
- [HE99] Minjie Hu and Stephen S. Easter. Retinal Neurogenesis: The Formation of the Initial Central Patch of Postmitotic Cells. *Developmental Biology*, 207(2):309–321, March 1999.
- [Hea67] D.F. Heath. Normal or Log-normal: Appropriate Distributions. *Nature*, 213:1159–1160, March 1967.
- [HH91] William A. Harris and Volker Hartenstein. Neuronal determination without cell division in xenopus embryos. *Neuron*, 6:499–515, 1991.
- [HHHL19] Edward Higson, Will Handley, Michael Hobson, and Anthony Lasenby. Dynamic nested sampling: An improved algorithm for parameter estimation and evidence calculation. *Statistics and Computing*, 29(5):891–913, September 2019.
- [HKB08] Herbert Hoijtink, Irene Klugkist, and Paul A. Boelen, editors. *Bayesian Evaluation of Informative Hypotheses*. Statistics for Social and Behavioral Sciences. Springer, New York, 2008.

- [HLZQ17] Sui Huang, Fangting Li, Joseph X. Zhou, and Hong Qian. Processes on the emergent landscapes of biochemical reaction networks and heterogeneous cell population dynamics: Differentiation in living matters. *Journal of the Royal Society Interface*, 14(130), May 2017.
- [Hor04] D. J. Horsford. Chx10 repression of Mitf is required for the maintenance of mammalian neuroretinal identity. *Development*, 132(1):177–187, December 2004.
- [HP98] William A. Harris and Muriel Perron. Molecular recapitulation: The growth of the vertebrate retina. *International Journal of Developmental Biology*, 42:299–304, 1998.
- [HSK⁺13] Steven A. Harvey, Ian Sealy, Ross Kettleborough, Fruzsina Fenyes, Richard White, Derek Stemple, and James C. Smith. Identification of the zebrafish maternal and paternal transcriptomes. *Development*, 140(13):2703–2710, July 2013.
- [HYSL11] Hongpeng He, Fa-Xing Yu, Chi Sun, and Yan Luo. CBP/p300 and SIRT1 Are Involved in Transcriptional Regulation of S-Phase Specific Histone Genes. *PLoS ONE*, 6(7):e22088, July 2011.
- [HZA⁺12] Jie He, Gen Zhang, Alexandra D. Almeida, Michel Cayouette, Benjamin D. Simons, and William A. Harris. How Variable Clones Build an Invariant Retina. *Neuron*, 75(5):786–798, September 2012.
- [ICW13] Kenzo Ivanovitch, Florencia Cavodeassi, and Stephen W. Wilson. Precocious Acquisition of Neuroepithelial Character in the Eye Field Underlies the Onset of Eye Morphogenesis. *Developmental Cell*, 27(3):293–305, November 2013.
- [IKRN16] Jaroslav Icha, Christiane Kunath, Mauricio Rocha-Martins, and Caren Norden. Independent modes of ganglion cell translocation ensure correct lamination of the zebrafish retina. *The Journal of Cell Biology*, 215(2):259–275, October 2016.
- [Ioa05] John P. A. Ioannidis. Why Most Published Research Findings Are False. *PLoS Medicine*, 2(8):e124, August 2005.
- [IYS⁺96] T Imai, M Yamauchi, N Seki, T Sugawara, T Saito, Y Matsuda, H Ito, T Nagase, N Nomura, and T Hori. Identification and characterization of a new gene physically linked to the ATM gene. *Genome Research*, 6(5):439–447, May 1996.
- [JBE03] Edwin T Jaynes, G. Larry Bretthorst, and EBSCO Publishing. *Probability Theory: The Logic of Science*. Cambridge University Press, Cambridge, 2003.
- [Kan06] Kunihiko Kaneko. *Life: An Introduction to Complex Systems Biology*. Understanding Complex Systems. Springer, Berlin ; New York, 2006.
- [Kes04] David Kestenbaum. The Not So Random Coin Toss. <https://www.npr.org/templates/story/story.php?storyId=1697475>, 2004.
- [KKT⁺13] Vijayalakshmi Kari, Oleksandra Karpiuk, Bettina Tieg, Malte Kriegs, Ekkehard Dikomey, Heike Krebber, Yvonne Begus-Nahrmann, and Steven A. Johnsen. A Subset of Histone H2B Genes Produces Polyadenylated mRNAs under a Variety of Cellular Conditions. *PLoS ONE*, 8(5):e63745, May 2013.

- [KKZ⁺18] Naeh L. Klages-Mundt, Ashok Kumar, Yuexuan Zhang, Prabodh Kapoor, and Xuetong Shen. The Nature of Actin-Family Proteins in Chromatin-Modifying Complexes. *Frontiers in Genetics*, 9, September 2018.
- [KMF⁺09] Noam Kaplan, Irene K. Moore, Yvonne Fondufe-Mittendorf, Andrea J. Gossett, Desiree Tillo, Yair Field, Emily M. LeProust, Timothy R. Hughes, Jason D. Lieb, Jonathan Widom, and Eran Segal. The DNA-encoded nucleosome organization of a eukaryotic genome. *Nature*, 458(7236):362–366, March 2009.
- [Kon06] Toru Kondo. Epigenetic alchemy for cell fate conversion. *Current Opinion in Genetics & Development*, 16(5):502–507, October 2006.
- [Kor05] Alfred Korzybski. *Science and Sanity: An Introduction to Non-Aristotelian Systems and General Semantics*. Inst. of General Semantics, Brooklyn, N.Y, 5. ed., 3. print edition, 2005.
- [Koz08] Zbynek Kozmik. The role of Pax genes in eye evolution. *Brain Research Bulletin*, 75(2-4):335–339, March 2008.
- [KSN99] Nathan L. Kleinman, James C. Spall, and Daniel Q. Naiman. Simulation-Based Optimization with Stochastic Approximation Using Common Random Numbers. *Management Science*, 45(11):1570–1578, 1999.
- [LAA⁺06] M. Locker, M. Agathocleous, M. A. Amato, K. Parain, W. A. Harris, and M. Perron. Hedgehog signaling and the retina: Insights into the mechanisms controlling the proliferative properties of neural precursors. *Genes & Development*, 20(21):3036–3048, November 2006.
- [Lar92] Ellen W. Larsen. Tissue strategies as developmental constraints: Implications for animal evolution. *Trends in Ecology & Evolution*, 7(12):414–417, December 1992.
- [LD09] Enhu Li and Eric H. Davidson. Building developmental gene regulatory networks. *Birth Defects Research Part C: Embryo Today: Reviews*, 87(2):123–130, June 2009.
- [LHKR08] Deepak A. Lamba, Susan Hayes, Mike O. Karl, and Thomas Reh. Baf60c is a component of the neural progenitor-specific BAF complex in developing retina. *Developmental Dynamics*, 237(10):3016–3023, September 2008.
- [LHO⁺00] Zheng Li, Minjie Hu, Małgorzata J. Ochocinska, Nancy M. Joseph, and Stephen S. Easter. Modulation of cell proliferation in the embryonic retina of zebrafish (*Danio rerio*). *Developmental Dynamics*, 219(3):391–401, 2000.
- [LV08] Ming Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Texts in Computer Science. Springer, New York, 3rd ed edition, 2008.
- [LWR⁺07] Julie Lessard, Jiang I. Wu, Jeffrey A. Ranish, Mimi Wan, Monte M. Winslow, Brett T. Staahl, Hai Wu, Ruedi Aebersold, Isabella A. Graef, and Gerald R. Crabtree. An Essential Switch in Subunit Composition of a Chromatin Remodeling Complex during Neural Development. *Neuron*, 55(2):201–215, July 2007.

- [Lyo13] Louis Lyons. Discovering the Significance of 5 sigma. *arXiv:1310.1284 [hep-ex, physics:hep-ph, physics:physics]*, October 2013.
- [Ma00] T. Ma. Cell cycle-regulated phosphorylation of p220NPAT by cyclin E/Cdk2 in Cajal bodies promotes histone gene transcription. *Genes & Development*, 14(18):2298–2313, September 2000.
- [MAA⁺01] Till Marquardt, Ruth Ashery-Padan, Nicole Andrejewski, Raffaella Scardigli, Francois Guillemot, and Peter Gruss. Pax6 Is Required for the Multipotent State of Retinal Progenitor Cells. *Trends in Biochemical Sciences*, 30(3):i, 2001.
- [MAB⁺13] Gary R. Mirams, Christopher J. Arthurs, Miguel O. Bernabeu, Rafel Bordas, Jonathan Cooper, Alberto Corrias, Yohan Davit, Sara-Jane Dunn, Alexander G. Fletcher, Daniel G. Harvey, Megan E. Marsh, James M. Osborne, Pras Pathmanathan, Joe Pitt-Francis, James Southern, Nejib Zemzemi, and David J. Gavaghan. Chaste: An Open Source C++ Library for Computational Physiology and Biology. *PLoS Computational Biology*, 9(3):e1002970, March 2013.
- [MBE⁺07] Karine Massé, Surinder Bhamra, Robert Eason, Nicholas Dale, and Elizabeth A. Jones. Purine-mediated signalling triggers eye development. *Nature*, 449(7165):1058–1062, October 2007.
- [MDBN⁺05] Juan-Ramon Martinez-Morales, Filippo Del Bene, Gabriela Nica, Matthias Hammerschmidt, Paola Bovolenta, and Joachim Wittbrodt. Differentiation of the Vertebrate Retina Is Coordinated by an FGF Signaling Center. *Developmental Cell*, 8(4):565–574, April 2005.
- [MFKM12] Kiran Mahajan, Bin Fang, John M Koomen, and Nupam P Mahajan. H2B Tyr37 phosphorylation suppresses expression of replication-dependent core histone genes. *Nature Structural & Molecular Biology*, 19(9):930–937, August 2012.
- [MGv⁺09] Partha Mitra, Prachi N. Ghule, Margaretha van der Deen, Ricardo Medina, Rong-lin Xie, William F. Holmes, Xin Ye, Keiichi I. Nakayama, J. Wade Harper, Janet L. Stein, Gary S. Stein, and Andre J. van Wijnen. CDK inhibitors selectively diminish cell cycle controlled activation of the histone H4 gene promoter by p220^{NPAT} and HiNF-P. *Journal of Cellular Physiology*, 219(2):438–448, May 2009.
- [MHR⁺99] Nicholas Marsh-Armstrong, Haochu Huang, Benjamin F Remo, Tong Tong Liu, and Donald D Brown. Asymmetric Growth and Development of the Xenopus laevis Retina during Metamorphosis Is Controlled by Type III Deiodinase. *Neuron*, 24(4):871–878, December 1999.
- [MMDM04] Kathryn B. Moore, Kathleen Mood, Ira O. Daar, and Sally A. Moody. Morphogenetic Movements Underlying Eye Field Formation Require Interactions between the FGF and ephrinB1 Signaling Pathways. *Developmental Cell*, 6(1):55–67, January 2004.
- [Mor03] Michel Morange. *La Vie Expliquée: 50 Ans Après La Double Hélice*. Sciences. O. Jacob, Paris, 2003.

- [Mor08] Michel Morange. What history tells us XIII. Fifty years of the Central Dogma. *Journal of biosciences*, 33(2):171–175, 2008.
- [Mor09] Michel Morange. A new revolution? The place of systems biology and synthetic biology in the history of biology. *EMBO Reports*, 10(Suppl 1):S50–S53, August 2009.
- [Mor11] Michel Morange. Recent opportunities for an increasing role for physical explanations in biology. *Studies in History and Philosophy of Biol & Biomed Sci*, 42(2):139–144, 2011.
- [Mun19] Bernard Munos. 2018 New Drugs Approvals: An All-Time Record, And A Watershed. *Forbes*, January 2019.
- [MXM⁺03] Partha Mitra, Rong-Lin Xie, Ricardo Medina, Hayk Hovhannisyan, S. Kaleem Zaidi, Yue Wei, J. Wade Harper, Janet L. Stein, André J. van Wijnen, and Gary S. Stein. Identification of HiNF-P, a Key Activator of Cell Cycle-Controlled Histone H4 Genes at the Onset of S Phase. *Molecular and Cellular Biology*, 23(22):8110–8123, November 2003.
- [MZLF02] A Murciano, J Zamora, J Lopez Sanchez, and J Frade. Interkinetic Nuclear Movement May Provide Spatial Clues to the Regulation of Neurogenesis. *Molecular and Cellular Neuroscience*, 21(2):285–300, October 2002.
- [Neu00] C. J. Neumann. Patterning of the Zebrafish Retina by a Wave of Sonic Hedgehog Activity. *Science*, 289(5487):2137–2139, September 2000.
- [NLM89] R. S. Nowakowski, S. B. Lewin, and M. W. Miller. Bromodeoxyuridine immunohistochemical determination of the lengths of the cell cycle and the DNA-synthetic phase for an anatomically defined population. *Journal of Neurocytology*, 18(3):311–318, June 1989.
- [NYLH09] Caren Norden, Stephen Young, Brian A. Link, and William A. Harris. Actomyosin Is the Main Driver of Interkinetic Nuclear Migration in the Retina. *Cell*, 138(6):1195–1208, September 2009.
- [PB04] David Posada and Thomas R. Buckley. Model Selection and Model Averaging in Phylogenetics: Advantages of Akaike Information Criterion and Bayesian Approaches Over Likelihood Ratio Tests. *Systematic Biology*, 53(5):793–808, October 2004.
- [PEM⁺09] David M. Parichy, Michael R. Elizondo, Margaret G. Mills, Tiffany N. Gordon, and Raymond E. Engeszer. Normal table of postembryonic zebrafish development: Staging by externally visible anatomy of the living fish. *Developmental Dynamics*, 238(12):2975–3015, December 2009.
- [PJ10] J. Pirngruber and S. A. Johnsen. Induced G1 cell-cycle arrest controls replication-dependent histone mRNA 3' end processing through p21, NPAT and CDK9. *Oncogene*, 29(19):2853–2863, 2010.
- [PKVH98] Muriel Perron, Shami Kanekar, Monica L. Vetter, and William A Harris. The genetic sequence of retinal development in the ciliary margin of the xenopus eye. *Developmental Biology*, (199):185–200, 1998.

- [PLM⁺17] Tao Peng, Linan Liu, Adam L MacLean, Chi Wut Wong, Weian Zhao, and Qing Nie. A mathematical model of mechanotransduction reveals how mechanical memory regulates mesenchymal stem cell fate decisions. *BMC Systems Biology*, 11, May 2017.
- [Poi13] Julia Pointner. *Genome-wide identification of nucleosome positioning determinants in Schizosaccharomyces pombe*. PhD thesis, Ludwig-Maximilians-Universität, München, July 2013.
- [PSS⁺09] Judith Pirngruber, Andrei Shchebet, Lisa Schreiber, Efrat Shema, Neri Minsky, Rob D Chapman, Dirk Eick, Yael Aylon, Moshe Oren, and Steven A Johnsen. CDK9 directs H2B monoubiquitination and controls replication-dependent histone mRNA 3'-end processing. *EMBO reports*, 10(8):894–900, August 2009.
- [RBBP06] Pamela A. Raymond, Linda K. Barthel, Rebecca L. Bernardos, and John J. Perkowski. Molecular characterization of retinal stem cells and their niches in adult zebrafish. *BMC developmental biology*, 6(1):36, 2006.
- [RDT⁺01] J. T. Rasmussen, M. A. Deardorff, C. Tan, M. S. Rao, P. S. Klein, and M. L. Vetter. Regulation of eye development by frizzled signaling in Xenopus. *Proceedings of the National Academy of Sciences*, 98(7):3861–3866, March 2001.
- [Res00] Nicholas Rescher. *Nature and Understanding*. Oxford University Press, New York, 2000.
- [Res05] Nicholas Rescher. *Cognitive Harmony*. University of Pittsburgh Press, Pittsburgh, PA, 2005.
- [RO04] Jonathan M. Raser and Erin K. O’Shea. Control of Stochasticity in Eukaryotic Gene Expression. *Science; Washington*, 304(5678):1811–4, June 2004.
- [Rv08] Arjun Raj and Alexander van Oudenaarden. Stochastic gene expression and its consequences. *Cell*, 135(2):216–226, October 2008.
- [Sad97] Payman Sadegh. Constrained Optimization via Stochastic Approximation with a Simultaneous Perturbation Gradient Approximation. *Automatica*, 33(5):889–892, May 1997.
- [Sch93] Kenneth F. Schaffner. *Discovery and Explanation in Biology and Medicine*. Science and Its Conceptual Foundations. University of Chicago Press, Chicago, 1993.
- [SF03] Deborah L Stenkamp and Ruth A Frey. Extraretinal and retinal hedgehog signaling sequentially regulate retinal differentiation in zebrafish. *Developmental Biology*, 258(2):349–363, June 2003.
- [SH14] Corinna Singleman and Nathalia G. Holtzman. Growth and Maturation in the Zebrafish, *Danio Rerio* : A Staging Tool for Teaching and Research. *Zebrafish*, 11(4):396–406, August 2014.
- [SK15] Zheng Sun and Natalia L. Komarova. Stochastic control of proliferation and differentiation in stem cell dynamics. *Journal of Mathematical Biology; Heidelberg*, 71(4):883–901, October 2015.

- [Ski06] John Skilling. Nested Sampling for Bayesian Computations. In *Proc. Valencia*, Benidorm (Alicante, Spain), June 2006. inference.org.uk.
- [Ski12] John Skilling. Bayesian computation in big spaces-nested sampling and Galilean Monte Carlo. In *BAYESIAN INFERENCE AND MAXIMUM ENTROPY METHODS IN SCIENCE AND ENGINEERING: 31st International Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering*, pages 145–156, Waterloo, Ontario, Canada, 2012.
- [Ski19] John Skilling. Galilean and Hamiltonian Monte Carlo. page 8, 2019.
- [Spa98] J. C. Spall. Implementation of the simultaneous perturbation algorithm for stochastic optimization. *IEEE Transactions on Aerospace and Electronic Systems*, 34(3):817–823, July 1998.
- [Ste18] Jacob Stegenga. *Medical Nihilism*. Oxford University Press, Oxford, United Kingdom, first edition edition, 2018.
- [STN⁺02] Masashi Sagara, Eri Takeda, Akiyo Nishiyama, Syunsaku Utsumi, Yoshiroh Toyama, Shigeki Yuasa, Yasuharu Ninomiya, and Takashi Imai. Characterization of functional regions for nuclear localization of NPAT. *The Journal of Biochemistry*, 132(6):875–879, 2002.
- [SVT13] Lourdes Serrano, Berta N Vazquez, and Jay Tischfield. Chromatin structure, pluripotency and differentiation. *Experimental Biology and Medicine*, 238(3):259–270, March 2013.
- [TC87] David L. Turner and Constance L. Cepko. A common progenitor for neurons and glia persists in rat retina late in development. *Nature*, 328(9), July 1987.
- [TCM⁺07] Michael Y. Tolstorukov, Andrew V. Colasanti, David M. McCandlish, Wilma K. Olson, and Victor B. Zhurkin. A Novel Roll-and-Slide Mechanism of DNA Folding in Chromatin: Implications for Nucleosome Positioning. *Journal of Molecular Biology*, 371(3):725–738, August 2007.
- [TK06] Dmitry N. Tsigankov and Alexei A. Koulakov. A unifying model for activity-dependent and activity-independent mechanisms predicts complete structure of topographic maps in ephrin-A deficient mice. *Journal of Computational Neuroscience*, 21(1):101–114, August 2006.
- [TMS64] J. E. Till, E. A. Mcculloch, and L. Siminovitch. A stochastic model of stem cell proliferation, based on the growth of spleen colony-forming cells. *Proceedings of the National Academy of Sciences of the United States of America*, 51:29–36, January 1964.
- [TR86] Sally Temple and Martin C. Raff. Clonal analysis of oligodendrocyte development in culture: Evidence for a developmental clock that counts cell divisions. *Cell*, 44(5):773–779, March 1986.
- [TR14] W.-W. Tee and D. Reinberg. Chromatin features and the epigenetic regulation of pluripotency states in ESCs. *Development*, 141(12):2376–2390, June 2014.

- [Tro00] V. Tropepe. Retinal Stem Cells in the Adult Mammalian Eye. *Science*, 287(5460):2032–2036, March 2000.
- [Tro08] Roberto Trotta. Bayes in the sky: Bayesian inference and model selection in cosmology. *Contemporary Physics*, 49(2):71–104, March 2008.
- [TSC90] D. L. Turner, E. Y. Snyder, and C. L. Cepko. Lineage-independent determination of cell type in the embryonic mouse retina. *Neuron*, 4(6):833–845, June 1990.
- [TSC08] Jeffrey M. Trimarchi, Michael B. Stadler, and Constance L. Cepko. Individual Retinal Progenitor Cells Display Extensive Heterogeneity of Gene Expression. *PLOS ONE*, 3(2):e1588, February 2008.
- [VGV⁺18] Jessie Van houcke, Emiel Geeraerts, Sophie Vanhunsel, An Beckers, Lut Noterdaeme, Marjijke Christiaens, Ilse Bollaerts, Lies De Groef, and Lieve Moons. Extensive growth is followed by neurodegenerative pathology in the continuously expanding adult zebrafish retina. *Biogerontology*, October 2018.
- [VJM⁺09] Marta Vitorino, Patricia R Jusuf, Daniel Maurus, Yukiko Kimura, Shin-ichi Higashijima, and William A Harris. Vsx2 in the zebrafish retina: Restricted lineages through derepression. *Neural Development*, 4(1):14, 2009.
- [VL54] V. Vilter and L. Lewis. [Existence and distribution of mitoses in the retina of the deepsea fish Bathylagus benedicti]. - PubMed - NCBI. *C R Seances Soc Biol Fil.*, 148(21-22):1771–5, 1954.
- [vM77] L. v. Salvini-Plawen and Ernst Mayr. On the Evolution of Photoreceptors and Eyes. In Max K. Hecht, William C. Steere, and Bruce Wallace, editors, *Evolutionary Biology*, pages 207–263. Springer US, Boston, MA, 1977.
- [Wag07] Günter P. Wagner. The developmental genetics of homology. *Nature Reviews Genetics*, 8(6):473–479, 2007.
- [WAR⁺16] Y. Wan, A. D. Almeida, S. Rulands, N. Chalour, L. Muresan, Y. Wu, B. D. Simons, J. He, and W. A. Harris. The ciliary marginal zone of the zebrafish retina: Clonal and time-lapse analysis of a continuously growing tissue. *Development*, 143(7):1099–1107, April 2016.
- [Wes00] M. Westerfield. The Zebrafish Book : A Guide for the Laboratory Use of Zebrafish. http://zfin.org/zf_info/zfbook/zfbk.html, 2000.
- [WF88] R. Wetts and S. E. Fraser. Multipotent precursors can give rise to all major cell types of the frog retina. *Science*, 239(4844):1142–1145, March 1988.
- [WG92] Robert W Williams and Dan Goldowitz. Lineage versus environment in embryonic retina: A revisionist perspective. *TINS*, 15(10):6, 1992.
- [WIEO04] Aiyan Wang, Tsuyoshi Ikura, Kazuhiro Eto, and Masato S. Ota. Dynamic interaction of p220NPAT and CBP/p300 promotes S-phase entry. *Biochemical and Biophysical Research Communications*, 325(4):1509–1516, December 2004.

- [Wik18] Wikipedia. Stochastic process. *Wikipedia*, July 2018.
- [Win15] R. Winklbauer. Cell adhesion strength from cortical tension - an integration of concepts. *Journal of Cell Science*, 128(20):3687–3693, October 2015.
- [WJH03] Y. Wei, J. Jin, and J. W. Harper. The Cyclin E/Cdk2 Substrate and Cajal Body Component p220NPAT Activates Histone Transcription through a Novel LisH-Like Domain. *Molecular and Cellular Biology*, 23(10):3669–3680, May 2003.
- [WR90] Takashi Watanabe and Martin C. Raff. Rod photoreceptor development in vitro: Intrinsic properties of proliferating neuroepithelial cells change as development proceeds in the rat retina. *Neuron*, 4(3):461–467, March 1990.
- [WR09] L. L. Wong and D. H. Rapaport. Defining retinal progenitor cell competence in *Xenopus laevis* by clonal analysis. *Development*, 136(10):1707–1715, May 2009.
- [WSM⁺05] Ann M. Wehman, Wendy Staub, Jason R. Meyers, Pamela A. Raymond, and Herwig Baier. Genetic dissection of the zebrafish retinal stem-cell compartment. *Developmental Biology*, 281(1):53–65, May 2005.
- [WSP⁺09] Minde I. Willardsen, Arminda Suli, Yi Pan, Nicholas Marsh-Armstrong, Chi-Bin Chien, Heithem El-Hodiri, Nadean L. Brown, Kathryn B. Moore, and Monica L. Vetter. Temporal regulation of Ath5 gene expression during eye development. *Developmental Biology*, 326(2):471–481, February 2009.
- [Yam05] M. Yamaguchi. Histone deacetylase 1 regulates retinal neurogenesis in zebrafish by suppressing Wnt and Notch signaling pathways. *Development*, 132(13):3027–3043, July 2005.
- [YDH⁺11] Jingye Yang, Huzefa Dungrawala, Hui Hua, Arkadi Manukyan, Lesley Abraham, Wesley Lane, Holly Mead, Jill Wright, and Brandt L. Schneider. Cell size and growth rate are major determinants of replicative lifespan. *Cell Cycle*, 10(1):144–155, January 2011.
- [You85] Richard W. Young. Cell differentiation in the retina of the mouse. *The Anatomical Record*, 212(2):199–205, June 1985.
- [YQW⁺18] Kai Yao, Suo Qiu, Yanbin V. Wang, Silvia J. H. Park, Ethan J. Mohns, Bhupesh Mehta, Xinran Liu, Bo Chang, David Zenisek, Michael C. Crair, Jonathan B. Demb, and Bo Chen. Restoration of vision after de novo genesis of rod photoreceptors in mammalian retinas. *Nature*, August 2018.
- [YSK15] Jienian Yang, Zheng Sun, and Natalia L. Komarova. Analysis of stochastic stem cell models with control. *Mathematical Biosciences*, 266(Supplement C):93–107, August 2015.
- [YWNH03] X. Ye, Y. Wei, G. Nalepa, and J. W. Harper. The Cyclin E/Cdk2 Substrate p220NPAT Is Required for S-Phase Entry, Histone Gene Expression, and Cajal Body Maintenance in Human Somatic Cells. *Molecular and Cellular Biology*, 23(23):8586–8600, December 2003.
- [ŽCC⁺05] Mihaela Žigman, Michel Cayouette, Christoforos Charalambous, Alexander Schleiffer, Oliver Hoeller, Dara Dunican, Christopher R. McCudden, Nicole Firnberg, Ben A. Barres, David P. Siderovski, and Juergen A. Knoblich. Mammalian Inscuteable Regulates Spindle

- Orientation and Cell Fate in the Developing Retina. *Neuron*, 48(4):539–545, November 2005.
- [ZDI⁺98] Jiyong Zhao, Brian Dynlacht, Takashi Imai, Tada-aki Hori, and Ed Harlow. Expression of NPAT, a novel substrate of cyclin E–CDK2, promotes S-phase entry. *Genes & development*, 12(4):456–461, 1998.
- [ZKL⁺00] Jiyong Zhao, Brian K. Kennedy, Brandon D. Lawrence, David A. Barbie, A. Gregory Matera, Jonathan A. Fletcher, and Ed Harlow. NPAT links cyclin E–Cdk2 to the regulation of replication-dependent histone gene transcription. *Genes & development*, 14(18):2283–2297, 2000.
- [ZMR⁺09] Yong Zhang, Zarmik Moqtaderi, Barbara P Rattner, Ghia Euskirchen, Michael Snyder, James T Kadonaga, X Shirley Liu, and Kevin Struhl. Intrinsic histone-DNA interactions are not the major determinant of nucleosome positions in vivo. *Nature Structural & Molecular Biology*, 16(8):847–852, August 2009.
- [Zub03] M. E. Zuber. Specification of the vertebrate eye by a network of eye field transcription factors. *Development*, 130(21):5155–5167, August 2003.